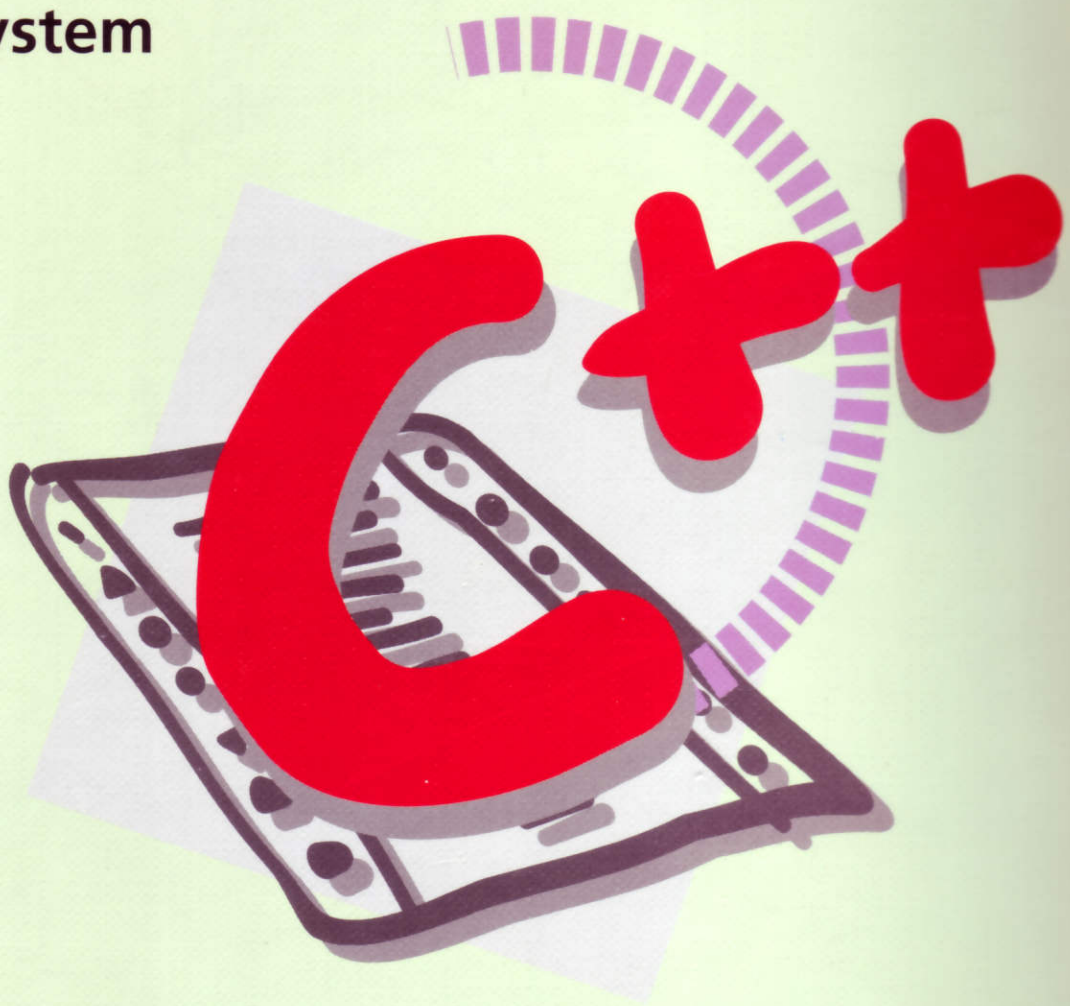
Maxon C++ 3

Integriertes Compilersystem



AMIGA®

MAXON computer

MaxonC++ Version 3.1

Wichtige Information: Bitte sofort lesen!

Sehr geehrter Kunde,

Sie haben von uns die neueste Version 3.1 von MaxonC++ erhalten. In dieser wurde der bisherige Editor *EDWARD* durch den neuen Editor *MaxonED* ersetzt, der wesentlich mehr Funktionen bietet, wie z.B. Farbhervorhebungen im Quelltext.

Durch diese Änderung sind die Teile des Handbuchs, in denen *EDWARD* behandelt wurde, überflüssig geworden. In Ihrem Handbuch fehlen dadurch die Seiten 85-156 und 187-200. Sie werden in den anderen Teilen des Handbuchs Verweise auf *EDWARD* finden, die allerdings größtenteils auch für *MaxonED* gelten. An der beschriebenen Zusammenarbeit zwischen Editor und Compiler hat sich prinzipiell nichts geändert.

Das Handbuch für *MaxonED* finden Sie nach der Installation von MaxonC++ als AmigaGuide-Datei im Verzeichnis „MCPP:BIN".

Es handelt sich dabei um eine Übergangslösung, die nur bis zum Erscheinen der Version 4 Ende 1996 beibehalten wird.

Senden Sie bitte unbedingt Ihre Registrierungskarte ein, damit wir Sie rechtzeitig über die neue Version und die Updatemöglichkeiten informieren können.

Ihre MAXON Computer GmbH

AMIGA

MaxonC++

Integriertes C++-Compilersystem

Developer Version

MAXON computer

MaxonC++ 3 Developer

Autoren:

Compiler:	**Jens Gelhar**
Editor:	**Thomas E. Wieger**
HotHelp 3:	**Hartmut Stein**
Easy-Objects:	**Jochen Becher**
Assembler:	**Klaus Dieter Sommer**
Debugger:	**Jochen Becher**

© 1992-95 MAXON Computer GmbH

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programmes, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitung enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **MaxonC++ 3** wenden Sie sich bitte an:

MAXON Computer GmbH
"**MaxonC++ 3** Support"
Postfach 5969
65734 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ ist ein Copyright der Commodore Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

AMIGA

MaxonC++

Integriertes C++-Compilersystem

MAXON
computer

MaxonC++ 3

Autoren:

Compiler:	**Jens Gelhar**
Editor:	**Thomas E. Wieger**
HotHelp 3:	**Hartmut Stein**
Easy-Objects:	**Jochen Becher**
Assembler:	**Klaus Dieter Sommer**
Debugger:	**Jochen Becher**

© 1992-94 MAXON Computer GmbH

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programmes, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitung enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **MaxonC++ 3** wenden Sie sich bitte an:

MAXON Computer GmbH
"**MaxonC++ 3** Support"
Postfach 5969
65734 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ ist ein Copyright der Commodore Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

MaxonC++ 3 ist da!

Nachdem die Welt der professionellen Softwareentwicklung die stille Revolution bereits erlebt hat, hat sich auch das Konzept der objektorientierten Programmierung sich in allen Bereichen durchgesetzt.

Nachdem Simula in den späten 60er Jahren ihrer Zeit zu weit voraus war und Smalltalk später zu sehr mit allen vertrauten Konzepten der Programmierung brach, um sich durchsetzen zu können, gibt es mit MaxonC++ eine objektorientierte Programmiersprache, die sich als der neue Standard in der Softwareentwicklung etabliert hat.

Es gibt viele Gründe, die gerade für C++ sprechen. Zunächst setzt diese Sprache auf C auf, der im professionellen Bereich bisher verbreitetsten Sprache. Das geht so weit, daß C-Quelltexte sich mit minimalen oder ganz ohne Modifikationen mit einem C++-Compiler übersetzen lassen. Diese Abwärtskompatiblität erleichtert natürlich den Umstieg von C nach C++ gewaltig und macht C++ für jeden C-Programmierer zur ersten Wahl. Ferner wurde C++ konsequent so konzipiert, daß eine objektorientierte Lösung weder beim Speicherplatz noch in Bezug auf die Laufzeit einer konventionellen Lösung unterlegen ist und so die Effizienz, die man von C gewohnt ist, erhalten bleibt.

Nicht zuletzt ist zu erwähnen, daß C++ von den AT&T Bell Laboratories entwickelt wurde und von AT&T, also gewiß keiner kleinen Firma, gefördert wird. Schon allein dieser Umstand hat für die Verbreitung von C++ im UNIX-Bereich und bei den sich mehr und mehr am Workstation-Markt orientierenden Microcomputer gesorgt.

Nur in der Amiga-Welt war von all diesen aufregenden Entwicklungen lange Zeit nichts zu spüren. Während auf anderen Systemen sogar jeder bessere C-Compiler objektorientierte Konzepte von C++ übernahm oder gleich zu einem vollständigen C++-System ausgebaut wurde, empfand der Amiga-User schon das Erscheinen von ANSI-kompatiblen C-Compilern als Fortschritt. Da leistungsfähige Compiler aber die Voraussetzung für professionelle Programmentwicklung sind, litt unter diesem Mangel nicht nur der persönliche Komfort der Amiga-Programmierer, sondern auch die Entwicklung der Amiga-Software allgemein wurde behindert und drohte den Anschluß an konkurrierende Systeme endgültig zu verlieren. Es war höchste Zeit, daß etwas geschah.

Nach mehr als dreijähriger Entwicklungszeit und einer umfangreichen Debugging-Phase ist MaxonC++ 3 jetzt einsatzbereit. MaxonC++ ist keine Portierung von einem anderen System, sondern wurde speziell für den Amiga (und auf einem Amiga) entwickelt und auf die Besonderheiten dieses Systems zugeschnitten. Der Compiler ist zum „3.0"-Standard (Light-Version zum 2.0-Standard) kompatibel und erzeugt aus dem C++-Quelltext direkt ausführbaren Code, ohne Umwege über C, Assembler oder andere Sprachen. Trotzdem läßt er sich auf ANSI-C-Emulation umschalten, um volle Kompatiblität zu den vorhandenen C-Compilern zu erreichen, und kann optional auch Assembler-Quelltext für die Weiterverarbeitung mit einem fremden Assemblern erzeugen.

Lieferumfang

Das MaxonC++ Compilersystem besteht aus zwei Handbüchern, die durch Einlageblätter voneinander getrennt sind:

1. MaxonC++ Anwender-Handbuch
   Compilersystem, Make, Linker, CLI-Verion, Editor und Anhänge
2. MaxonC++ Tutorial – Einführung in C und C++

Die Developer-Version des Compilersystems ist umweitere Handbücher ergänzt:

3. Tutorial-Ergänzung – Exceptions, Templates
4. MaxonC++ Referenz-Handbuch
5. Maxon HotHelp 3 – Online-Hilfe
6. MaxonEASY-OBJECTS – Klassenbibliothek
7. Maxon Source-Level-Debugger
8. MaxonASM (CLI-Version)

Der Compiler und die Zusatzprogramme werden auf mehreren Disketten geliefert. Genaue Angaben zur Installation und zur Diskettenanzahl entnehmen Sie bitte der „Readme"-Datei auf der Installationsdiskette. Da sich nach der Drucklegung noch einige Änderungen, was die Diskettenanzahl und die Programmzusammenstellung angehen, ergeben können, ist dies der beste Weg, um das Handbuch so aktuell wie möglich zu halten.

Developer-Upgrade

Allen registrierten Besitzern der Light-Version bieten wir die Möglichkeit eines Upgrades auf die MaxonC++-Developer-Version. Das Upgrade beinhaltet ein neues Compilersystem, die Klassenbibliothek, den MaxonSource-Level-Debugger, die Online-Hilfe HohHelp 3 und den MaxonASM in der CLI-Verion. Die Upgradekonditionen erfragen Sie bitte bei Maxon Computer: 06196/481811.

Registrierung

Ihrem Compilersystem liegt eine Registrierkarte bei, die Sie umgehend an uns zurücksenden sollten. Sie gewährleisten dadurch, daß Sie regelmäßig über Produktneuerungen und Neuerscheinungen informiert werden. Desweiteren steht Ihnen dann unsere tägliche Hotline für Problemlösungen zur Verfügung.

Hotline und Support

Unsere Hotline ist werktags zwischen 15:00 und 17:00 Uhr unter der Rufnummer 06196/43780 zu erreichen. Bitte prüfen Sie vor einem Anruf genau den Problemfall:

- Haben Sie die Software richtig installiert?
- Benutzen Sie die richtige Workbench Version (nicht unter 2.0)?
- Laufen irgendwelche Programme im Hintergrund (Screenblanker, Maustools o.ä.)?

Bitte haben Sie Verständnis, daß wir keine Garantie für das Zusammenspiel mit anderer Software geben können. Besonders einige Tools, die in der Startup-Sequence mitgestartet werden, können die Arbeitsfähigkeit sehr stören.

Bei einem Anruf halten Sie bitte Ihre Registriernummer, die Versionsnummer des jeweiligen Softwarepaketes und eine genaue Konfigurationsbeschreibung bereit. Die Registriernummer ist auf den Orginal-Disketten aufgedruckt. Die Versionsnummer finden Sie entwender im Info-Dialogfenster des jeweiligen Programmes oder wenn Sie (ab Amiga OS 2.0) den Befehl:

Version "Programmname" full

in der Shell (CLI) eingeben.

Wenn Sie uns schreiben, benötigen wir zum zügigen Bearbeiten, ebenfalls die beschriebenen Angaben.

Richten Sie Ihr Anschreiben an folgende Adresse:

Maxon Computer GmbH
"MaxonC++ 3-Support"
Industriestraße 26
W-65760 Eschborn

Version
3.10 r1 (22.05.96)

Literaturangaben

Wenn Sie bereits in einer anderen Programmiersprache als C++ programmieren können, ist das Tutorial-Handbuch eine sehr gute Einführung für Sie, um die Syntax und den Umgang mit C++ kennenzulernen. Als Programmieranfänger empfiehlt es sich jedoch, ein Grundeinführungsbuch durchzuarbeiten. Die nachfolgende Liste enthält eine Auswahl an Büchern zu den Themen objektorientierte Programmierung im allgemeinen, Programmierung in C und C++ und zur Programmierung des Amiga Betriebssystems.

Objektorientierte Programmierung allgemein

Objektorientierte Programmierung - Eine Einführung
R. Engel - 338 Seiten - ISBN 3-89090-353-3
Markt&Technik Verlag AG/Haar

C und C++ Literatur
Lernen und Üben: C-Schulung
H. Niegemann und T. Schäfer - 354 Seiten - ISBN 3-89090-218-9
Markt&Technik Verlag AG/Haar

Die Einführung: C
S. Baloui - 379 Seiten - ISBN 3-89090-334-7
Markt&Technik Verlag AG/Haar

Programmieren in C++
E. Huckert - 348 Seiten - ISBN 3-89090-171-9
Markt&Technik Verlag AG/Haar

Clever Programmieren: C++
M. Haun - 352 Seiten - ISBN 3-88322-327-1
IWT Verlag GmbH/Vaterstetten

IWT Kompakt: C-Referenz für alle ANSI-C-Compiler
P. Prinz - 304 Seiten - ISBN 3-88322-340-9
IWT Verlag GmbH/Vaterstetten

C++ lernen und beherrschen: Einführung und Leitfaden
S. B. Lippmann - ca. 500 Seiten - ISBN 3-89319-375-8
Addison-Wesley (Deutschland) GmbH/Bonn

Die C++ Programmiersprache
B. Stroustrup - 380 Seiten - ISBN 3-89319-386-3
Addison-Wesley (Deutschland) GmbH/Bonn

The C++ Programming Language
B. Stroustrup - 576 pages - ISBN 0-201-53992-6
Addison-Wesley (Deutschland) GmbH/Bonn

The Annotated C++ Reference Manual
M. A. Ellis/ B. Stroustrup - 400 pages - ISBN 0-201-51459-1
Addison-Wesley (Deutschland) GmbH/Bonn

Die Kunst der objektorientierten Programmierung - Prinzipien und Konzepte in C++
A. Schäpers - ca. 440 Seiten - ISBN 3-89319-226-3
Addison-Wesley (Deutschland) GmbH/Bonn

C-Programmieren: Ein Kurs zum Selbststudium
G. Krüger - ca. 500 Seiten - ISBN 3-89319-372-3
Addison-Wesley (Deutschland) GmbH/Bonn

Der C-Experte: Programmieren ohne Pannen
A. Koenig - 190 Seiten - ISBN 3-89319-233-6
Addison-Wesley (Deutschland) GmbH/Bonn

Debugging C
R. Ward - 322 Seiten - ISBN 3-925118-90-X
Addison-Wesley (Deutschland) GmbH/Bonn

C Professional Edition
W. Herglotz - ca 450 Seiten - ISBN 3-89360-004-3
BHV Verlags GmbH/Korschenbroich

Das Einsteigerseminar C
W. Herglotz - 224 Seiten - ISBN 3-89360-606-8
BHV Verlags GmbH/Korschenbroich

Das Einsteigerseminar C++
W. Herglotz - 224 Seiten - ISBN 3-89360-622-X
BHV Verlags GmbH/Korschenbroich

C++ für Programmierer
F. Bause/W. Tölle - 286 Seiten - ISBN 3-528-05110-8
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

Einführung in die Programmiersprache C++
F. Bause/W. Tölle - 171 Seiten - ISBN 3-528-14689-3
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

Referenzhandbuch Standart C
P.J. Plauger/J. Brodie - 236 Seiten - ISBN 3-528-04741-0
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

C++ Einführung in die objektorientierte Programmierung
S. Prata - 750 Seiten - ISBN 3-89362-701-4
tewi Verlag

Der schnelle Umstieg auf C++
M. Aupperle - 76 Seiten (Chip Spezial) - ISBN 3-8023-1088-8
Vogel Verlag und Druck KG/Würzburg

C für Umsteiger
B. Klöppel - 122 Seiten (Chip Spezial) - ISBN 3-8023-1081-0
Vogel Verlag und Druck KG/Würzburg

Amiga Bücher

Amiga-Programmierrichtlinien
H. Gzella - 272 Seiten - ISBN 3-87791-049-1
Markt&Technik Verlag AG/Haar

Amiga ROM Kernel Reference Manual: Devices
512 pages - ISBN 0-201-56775-X
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Libraries
960 pages - ISBN 0-201-56774-1
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Includes and Autodocs
1000 pages - ISBN 0-201-56773-3
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga Hardware Reference Manual
512 pages - ISBN 0-201-56776-8
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga User Interface Style Guide
224 pages - ISBN 0-201-57757- 7
Addison-Wesley (Deutschland) GmbH/Bonn

MaxonHotHelp 3 OS 3.1 Dokumentationen
Deutschsprachige Online-Dokumentation Best.Nr.: 920313
Beschreibt alle Betriebssystemfunktionen unter Kickstart/Workbench 3.1
MAXON Computer GmbH Eschborn

Anwenderhandbuch

MaxonC++ 3 ist da! V-5

Lieferumfang V-6
Developer-Upgrade V-6
Registrierung V-6
Hotline und Support V-6
Literaturangaben V-8
Objektorientierte Programmierung allgemein V-8
Amiga Bücher V-11

Maxon C++ 3 3

1.1 Was gibt es Neues? 3
1.2 Der Umstieg 4
1.2.1 Konfigurationsdateien 4
1.2.2 „Edward.DEF" 4
1.2.3 Dateien und Verzeichnisse 6
1.3 Ein ersterÜberblick 7
1.3.1 Disketten 7
1.3.2 Installation 7
1.3.3 Die Komponenten des MaxonC++ Systems 7
1.4 Erste Schritte mit MCPP 9

2. Die integrierte Entwickungsumgebung 11

2.1 Programmstart und Optionen 11
2.2 Compiler und Editor - eine harmonische Beziehungskiste 12
2.3 Das Konzept 13
2.4 Compiler-Funktionen 14
2.4.1 Programme übersetzen 14
2.4.1.1 Das Compiler-Fenster 14
2.4.1.2 Abbruch und Ausstieg 16
2.4.1.3 Auswahl von Fehlermeldungen 16
2.4.1.4 Vom Compiler in den Editor 17
2.4.1.5 „Weiter" 18

2.4.1.6 Programmstart aus dem Compiler-Fenster 18
2.4.1.7 Erneutes öffnen des Compiler-Fenster 18
2.4.2 Ausführen und Abbrechen 18
2.4.3 Aufruf des Debuggers 19
2.4.4 Exe- und Objektdateien 19
2.4.5 Daten und Informationen 20
2.5 Features und Optionen 20
2.5.1 Ein kurzer Überblick 20
2.5.2 Compiler-Optionen 20
2.5.2.1 Include-Dateien 20
2.5.2.2 Vorcompilierte Header-Dateien 22
2.5.2.3 Der Arbeitsspeicher des Compilers 23
2.5.2.4 Debug-Datei 23
2.5.2.5 Vordefinieren von Symbolen 23
2.5.2.6 Diverse kleine, aber trotzdem wichtige Schalter 24
2.5.2.6.1 C++ 24
2.5.2.6.2 Unterbrechen 24
2.5.2.6.3 IntMult 32 24
2.5.2.6.4 Debug-Datei 25
2.5.2.6.5 Symbol-Hunk 25
2.5.2.6.6 Asm schreiben 25
2.5.2.6.7 Source in Asm 26
2.5.2.6.8 Optimieren 26
2.5.2.6.9 Code für höhere Prozessorgenerationen 26
2.5.2.6.10 Strategien für Template-Funktionen 27
2.5.2.6.11 Exceptions 28
2.5.2.7 Optionale Warnungen und Meldungen 28
2.5.3 Der Linker-Requester 30
2.5.3.1 Datenmodelle 30
2.5.3.2 Linken oder nicht Linken - das ist hier die Frage 31
2.5.3.3 Verhalten bei Fehler 32
2.5.3.4 Bibliotheken 32
2.5.3.5 Zusammenfassen von Hunks 32
2.5.3.6 Schreiben von Dateien 33
2.5.4 MCPP-Systemeinstellungen 34
2.5.4.1 Erzeugen von Icons 34
2.5.4.2 Abhängigkeiten in Make-Dateien 34
2.5.4.3 Protokollieren von Ein- und Ausgaben 34
2.5.4.4 Argumente aus der imaginären Kommandozeile 34
2.5.4.5 Einbindung des Maxon Assemblers 35

2.5.4.6 Debugger 35
2.5.4.7 Stackgröße 35
2.5.4.8 Autosave 35
2.5.4.9 Dateiauswahlbox 36
2.5.5 Abspeichern der Konfigurationsdatei 36
2.5.5.1 Allgemeines 36
2.5.5.2 Der Editor 36
2.5.5.3 Fensterdaten 37
2.5.5.4 Switches und Schalter 37
2.5.5.5 Stringoptionen 37
2.6 Die ARexx-Schnittstelle 38
2.6.1 Grundlagen 38
2.6.2 Der ARexx-Port 38
2.6.3 Die ARexx-Befehle 39
2.6.3.1 Schließen, öffnen und Beenden 39
2.6.3.2 Übersetzen, Starten und Ähnliches 39
2.6.3.3 Requester und Informationen 40
2.6.3.4 Make 40
2.6.3.5 Schreiben von Dateien 40
2.6.3.6 Optionen speichern und setzen 40
2.6.3.7 Informationen über die Entwicklungsumgebung 41
2.6.3.8 Fehlermeldungen und Fehlerstellenansprung 41
3. MAKE - eine integrierte Projektverwaltung 43
3.1 Das Prinzip 43
3.1.1 Wozu eine Projektverwaltung? 43
3.1.2 Was ist das überhaupt - ein Projekt? 44
3.1.3 Was hat das ganze mit „Make" zu tun? 45
3.2 Die Praxis 45
3.2.1 Erstellen von Projekten 45
3.2.2 Hinzufügen von Dateien, umständliche Methode 46
3.2.3 Übersetzen des Projekts 48
3.2.4 Hinzufügen von Dateien, einfache Methode 48
3.2.5 Abhängigkeiten 48
3.2.6 Weitere Funktionen im Make-Fenster 49
3.2.6.1 Projektpflege 49
3.2.6.2 Übersetzung erzwingen 50
3.2.6.3 Dateien in den Editor laden 50
3.2.6.4 Projektdatei wechseln 50

3.2.7 Compileranweisungen für Objektdateien 50
3.2.7.1 Interne Übersetzung 50
3.3 Handgemachte Makedateien 52
3.3.1 Die Grundlagen 52
3.3.2 Der Linkeraufruf 52
3.3.3 Internes Compilieren 53
3.3.4 Externe Übersetzung 53
3.3.5 Weitere Features 54
3.3.5.1 Zeilenverbindung 54
3.3.5.2 Kommentare 54
3.3.5.3 Makros 55
3.3.5.4 Regeln 56
4.1 Prinzipielles 59
4.2 Standard-Funktionen und das "Logfile" 60
4.3 Das Hauptprogramm und seine Argumente 62
4.4 Linker-Bibliotheken 63
4.5 Initialisierungen und das große Aufräumen 64
4.6 Mit und ohne Startup-Code 65
4.7 Der Startup-Code 67
4.8 Hunks 68
5.1 Was ist neu? 69
5.2 Das Konzept 70
5.3 Optionen 71
5.3.1 Allgemeines 71
5.3.2 Ergebnisse und Ausgaben 72
5.3.2.1 Assembler-Quelltext: -a und -as 72
5.3.2.2 Objektdateien: -c 73
5.3.2.3 Ausführbare Programmdatei: -o 73
5.3.3 Preprozessor 73
5.3.3.1 Definieren von Symbolen: -d 73
5.3.3.2 Includedateien: -i 74
5.3.4 Compiler-Modus 74
5.3.4.1 ANSI C oder C++: -pc und -pp 74
5.3.4.2 Exception Handling: -px, -pX und -X 74
5.3.4.3 Erzeugen von Template-Funktionen: -t und -T 75
5.3.5 Codeerzeugung 75

5.3.5.1 Breakpoints setzen: -gb 75
5.3.5.2 Datenmodell: -gd und -gD 75
5.3.5.3 Zusammenfassen von Hunks: -gh 75
5.3.5.4 Ganzzahlige Arithmetik: -gm 75
5.3.5.5 Optimierungen: -go und -gO 76
5.3.5.6 Code für besondere Prozessoren: -g00, -g20, -g80 und -g81 76
5.3.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw 76
5.3.5.7 Code für Link-Libraries: -gl, -gL und -L 77
5.3.6 Debugger 77
5.3.6.1 Maxon Debugger: -b, -bi und -b0 77
5.3.6.2 Symbolischer Debugger: -bs und -bs0 77
5.3.7 Warnungen und Fehler 77
5.3.7.1 Format der Fehlerausgabe: -e, -el und -ei 77
5.3.7.2 Farben und Stile: -eb, -eB und -ec 78
5.3.7.3 Fehlerdatei: -ef 78
5.3.7.4 Optionale Warnungen: -w 78
5.3.7.5 Warnungen wie Fehler behandeln: -ew und -eW 79
5.3.8 Linker 79
5.3.8.1 Library-Pfade: -j 79
5.3.8.2 Linker-Libraries: -l 79
5.3.9 Die integrierte Projektverwaltung: -m, -mx und -mX 80
5.3.10 Arbeitsspeicher: -w 80
5.3.11 Vorcompilierte Headerdateien: -H, -h und -h0 81
5.4 Voreinstellungen mit "CPPOPT" 81
5.5 Zusammenfassung 82
6.1 Installation von Edward auf der Festplatte 85
6.2 Aufruf von Edward 85
6.2.1 Start von der Workbench 85
6.1.2 Start aus dem CLI 86
6.2 Eingabe und Bearbeitung von Texten 86
6.2.1 Die Status-Anzeige 86
6.2.2 BACKSPACE und DELETE 87
6.2.3 UndoLine-Funktion 87
6.2.4. RETURN, ENTER und TAB 87
6.2.5 Die Cursor-Tasten und der Ziffernblock 88
6.2.6 Der Schieberegler 88
6.2.7 Die Maus 88
6.3 Text 89

6.4.1 Erzeugen von Texten 89
6.3.2 Wahl eines bestimmten Textes 89
6.3.3 Freigeben eines Textes 90
6.3.4 Löschen des aktuellen Textes 90
6.3.5 Drucken eines Textes 90
6.4 Dateioperationen 91
6.4.1 Speichern von Texten 91
6.4.2 Laden von Texten 92
6.5. Blockoperationen 92
6.5.1 Cut, Copy, Paste 92
6.5.2 Spaltenblöcke 94
6.5.3 Laden, Speichern und Drucken 94
6.5.4 Einrücken 95
6.5.5 Umwandeln in Groß- bzw. Kleinbuchstaben 95
6.6 Suchen und Ersetzen 96
6.6.1 Das „Find"-Formular 96
6.6.2. Das „Find & Replace"-Formular 97
6.6.3 Struktur-Check (Klammer-Check) 98
6.6.4 Compiler Error-Datei interpretieren, Fehler anspringen 99
6.6.5 Auf bestimmte Zeilen positionieren 99
6.7 Strukturieren von Texten, Lesezeichen 100
6.7.1 Falten 100
6.7.2 Auszeichnen von Zeilen 102
6.7.3 Cursor-Bewegung 102
6.7.4 Lesezeichen (Bookmarks) 102
6.9 „Word-Wrap" 104
6.10 Makros 105
6.10.1 Makros aufzeichnen & abspielen 105
6.10.2 Makros speichern & abspielen 106
6.11 Fenster 107
6.11.1 Erzeugen von Fenstern 107
6.11.2 Wahl eines bestimmten Fensters 107
6.11.3 Schließen von Fenstern 108
6.11.4 Arbeiten mit mehreren Ansichten eines Textes 109
6.12 Verlassen des Editors 110
6.12.1. Unterbrechen der Arbeit 110
6.12.2 Verlassen des Editors 111

6.13 Iconify & HotKey 111
6.13.1 Iconify 111
6.13.2 HotKey 111
6.14 Der Kommandozeilen-Interpreter 112
6.15 Globale Einstellungen 112
6.15.1 Das „Globale Einstellungen"-Formular 112
6.15.2 Das „Seitenformat"-Formular 116
6.16 Benutzerdefinierte Menüs, Tastenbelegung, HotKey 117
6.16.1 Kurze Einführung in die EBNF-Darstellung 118
6.16.2 Definition eigener Menüs 119
6.16.3 Definition der Tastaturbelegung 120
6.16.4 Definition des HotKeys 123
6.16.5 Einladen einer neuen Definition 123
6.17 ARexx - Kommunikation mit Edward 123
6.18 Zurücknehmen von Text-Änderungen 124

Die Kommandos des Editors 126

1 Eingabe und Bearbeitung von Texten 127
2 Verwalten von Texten 131
3 Dateioperationen 133
4 Blockoperationen 134
5 Suchen und Ersetzen 137
6 Strukturieren von Texten 139
7 Ändern von Optionen für den aktuellen Text 142
8 Word-Wrap 143
9 Makros 144
10 Fenster 145
11 Verlassen des Editors 146
12 Formulare 147
13 Der Kommandozeilen-Interpreter 148
14 Ändern von globalen Einstellungen 149
15 ARexx - Kommunikation mit Edward 151
15.1 Abfrage von Statuswerten Edwards : 151

15.2 Laden von Texten aus Edward heraus 154
16 UnDo/ReDo - Zurücknehmen von Textänderungen 155
17 Verschiedenes 156

Anhang A 157
Veränderungen seit der Version 1.0 157
A-1. Compiler 157
A-1.1 Preprozessor 157
A-1.2 Parser und Sprachstandard 157
A-1.3 Codeerzeugung 158
A-1.4 Linker 158
A-2. Bibliotheken 159
A-3. Sonstiges 159
B-1. Warnungen 160
B-2. Fehlermeldungen 162
B-3. Sonstige Fehlermeldungen 185

Anhang-C – Editor-Fehlermeldungen 187
C-1 Informationen 187
C-2 Allgemeine Fehlermeldungen 187
C-3 Fehlermeldungen mit Abfragen 191
C-4 Meldungen mit Abfragen 191
C-5 Fehler bei Betriebssystemoperationen 193
C-6 Eingabefehler 195
C-7 Fehler in den Definitionsdateien 196
C-8 Fehler beim Analysieren von Kommandos 197
C-9 Fehlermeldungen beim Auswerten von Kommandos 199

Tutorial

Einführung in C++ 203
Ein paar kurze Bemerkungen, bevor es los geht 203

1. Der Einstieg 204
1.1 Ein erstes Beispiel 204
1.2 Lexikalisches 206
1.3 Zahlen, Variablen und Berechnungen 208
1.4 Numerische Datentypen und Operatoren 210
1.4.1 Ganzzahlige Datentypen 210
1.4.2 Literale für ganze Zahlen und Zeichenketten 212
1.4.3 Ganzzahlige Operatoren und Berechnungen 214
1.4.3.1 Allgemeines 214
1.4.3.2 Postfix-Operatoren 215
1.4.3.3 Unäre Operatoren 215
1.4.3.4 Multiplikative Operatoren 217
1.4.3.5 Addition und Subtraktion 218
1.4.3.6 Shift-Operatoren>> 218
1.4.3.7 Vergleiche 220
1.4.3.8 Bitweise Verknüpfungen 221
1.4.3.9 Logische Operatoren 221
1.4.3.10 Der bedingte Ausdruck 222
1.4.3.11 Zuweisungen 223
1.4.3.12 Der Komma-Operator 224
1.4.4 Fließkommatypen 224
1.4.4.1 Literale und Typen 224
1.4.4.2 Interne Darstellung der Daten 225
1.4.4.2 Fließkommaoperationen 226
1.4.5 Auswertung von Ausdrücken 226
1.4.6 Explizite Typumwandlungen 228
1.4.7 Initialisierungen 229
1.5 Anweisungen 230
1.5.1 Bedingungen und Verzweigungen 230
1.5.2 Anweisungen, Deklarationen und Blöcke 232
1.5.3 Schleifen 235
1.5.3.1 Die „while"-Anweisung 235
1.5.3.2 Die „for"-Schleife 236

1.5.3.3 Die „do"-Schleife 237
1.5.4 Sprünge 238
1.5.4.1 Sinn und Unsinn von „goto" 238
1.5.4.2 „break" und „continue" 240
1.5.5 Vielfachverzweigungen 242
1.5.6 Die „return"-Anweisung 244
1.6 Funktionen 244
1.6.1 „main" allein genügt nicht 244
1.6.2 Parameter und Argumente 245
1.6.3 Rückgabe von Werten 247
1.6.4 Funktionen und Gültigkeitsbereiche 249
1.6.4.1 Deklarationen auf Dateiebene 249
1.6.4.2 Rekursive Funktionen 251
1.6.4.3 Deklarationen und Definitionen 251
1.6.5 Überladen, Default-Parameter und mehr 253
1.6.5.1 Überladen von Funktionen 253
1.6.5.2 Abkürzen durch Default-Argumente 255
1.6.5.3 Variable Parameterlisten 257
1.6.6 Anachronismen aus der Mottenkiste 258
1.6.6.1 Umschalten in den C-Modus 258
1.6.6.2 Parameterlisten im alten Stil 259
1.6.6.3 Von der Steinzeit in die ANSI-Ära 260
1.6.7 Einige Standardfunktionen 261
1.6.7.1 Allgemeines 261
1.6.7.2 Ein- und Ausgabe: <stdio.h> 262
1.6.7.3 Zeichen, Zeichenketten und Bytefolgen: 262
1.6.7.4 Fließkommafunktionen: <math.h> 263
1.6.7.5 Die C-Fundgrube: <stdlib.h> 263

2. Datentypen und Deklarationen 265
2.1 Vektoren, Zeiger und Referenzen 265
2.1.1 Pointer 265
2.1.1.1 Von Objekten und Adressen 265
2.1.1.2 Zeiger: Deklaration und elementare Operationen 265
2.1.1.3 Zeiger als Funktionsparameter 267
2.1.1.4 Zeigertypen 268
2.1.1.5 Was man sonst noch so mit Zeigern machen kann 270
2.1.2 Vektoren 271
2.1.2.1 Das Vektor-Konzept in C 271
2.1.2.2 Zeiger und Vektoren 272

2.1.2.3 Genaueres über Vektordeklarationen 275
2.1.2.4 Vektoren als Parameter 277
2.1.2.5 Strings 278
2.1.2.6 Initialisierung von Vektoren 286
2.1.3 Referenzen 288
2.1.3.1 Einführung 288
2.1.3.2 „VAR-Parameter" und temporäre Objekte 289
2.1.3.3 Weitere Möglichkeiten und Einschränkungen 291
2.2 Qualifizierungen, Deklarationen und Spezifikationen 292
2.2.1 Speicherklassen 292
2.2.1.1 Automatische und statische Daten 292
2.2.1.2 Module und ihre Gültigkeitsbereiche 294
2.2.1.3 Daten in Registern 301
2.2.1.4 Daten im CHIP-RAM 303
2.2.2 Typdefinitionen 304
2.2.3 Konstante Daten auf der Flucht 305
2.2.3.1 Die „const"-Qualifizierung 305
2.2.3.2 Flüchtige Daten: „volatile" 308
2.2.4 Inline-Funktionen 309
2.3 Zeiger auf Funktionen 311
2.3.1 Wohin zeigen Zeiger auf Funktionen? 311
2.3.2 Anmerkungen zu Syntax und anderen Details 312
2.3.3 Zeiger auf Überladene Funktionen 313
2.3.4 Eine Anwendung: „qsort" und „bsearch" 314
2.4 Aufzählungen 316
2.4.1 Deklaration 316
2.4.2 Typnamen und Typlabel 318
2.4.3 Aufzählungen und Zahlen 319
2.5 Strukturen 321
2.5.1 Grundlagen 321
2.5.2 Definitionen und Deklarationen 323
2.5.3 Initialisierung 325
2.5.4 Gültigkeitsbereiche und Scope-Regeln 326
2.5.5 Unionen 328
2.5.6 Bitfelder 332
2.6 Dynamisch organisierte Datenstrukturen 333
2.6.1 Die Idee 333
2.6.2 „new" und „delete" 334

2.6.3 Dynamisch erzeugte Vektoren 335
2.6.4 Initialisierungen 336
2.6.5 „malloc" und andere Antiquitäten 337
2.6.6 Beispielprogramm: Eine Liste von Daten 339
2.6.6.1 Das Problem: Ein wirrer Haufen von Daten 339
2.6.6.2 Die Lösung: Eine doppelt verkettete Liste 341
2.6.6.3 Initialisieren und Einfügen 343
2.6.6.4 Suchen und Löschen 346
2.6.6.5 Das Programm an einem Stück 348

3. Objektorientiertes Programmieren 353

3.1 Klassen und Objekte 353
3.1.1 Keine Panik! 353
3.1.2 Member-Funktionen 355
3.1.2.1 Erste Beispiele 355
3.14.2.2 Member-Funktionen (Methoden) und konstante Objekte 357
3.1.2.3 Inlining von Member-Funktionen 357
3.1.3 Klassen und Zugriffsrechte 358
3.1.4 No Scope, no hope! 360
3.1.5 Friends 362
3.1.6 Statische Member 364

3.2 Vererbungslehre 366
3.2.1 Abgeleitete Klassen 366
3.2.1.1 Ein erstes Beispiel 366
3.2.1.2 Mehrfach-Vererbung und andere Features 369
3.2.1.3 Vererbung und Typregeln 371
3.2.2 Scoperegeln und Zugriffskontrolle bei Vererbung 373
3.2.2.1 Noch mehr Details 373
3.2.2.2 Der Zugriffsmodus „protected" 374
3.2.2.3 Zugriffs-Deklarationen 375
3.2.3 Virtuelle Member-Funktionen 377
3.2.3.1 Ein Beispiel 377
3.2.3.2 Die Details 379
3.2.3.3 Abstrakte Klassen 381
3.2.4 Ein paar Anmerkungen zur Implementation 382
3.2.5 Virtuelle Basisklassen 385

3.3 Konstruktoren und Destruktoren 393
3.3.1 Initialisierung von Klassen 393
3.3.2 Konstruktoren mit Argumenten 394

3.3.3 Initialisierung von Basisklassen und Membern 396
3.3.4 Besondere Konstruktoren 400
3.3.4.1 Default-Konstruktoren 400
3.3.4.2 Kopier-Konstruktoren 401
3.3.5 Destruktoren 403
3.4 Beispielprogramm: 406
3.5 Noch mehr Features 420
3.5.1 Zeiger auf Member 420
3.5.2 Konvertierungsfuntionen 423
3.5.3 Temporäre Objekte 426
4. Überladen 429
4.1 Überladene Funktionen 429
4.1.1 Was man darf, und was nicht 429
4.1.2 Was der Compiler davon hält 430
4.1.2.1 Die Hierarchie der Typwandlungen 430
4.1.2.2 Matching von Funktionsargumenten 435
4.2 Operatoren 437
4.2.1 Allgemeine Operatoren 437
4.2.1.1 Allgemeines über allgemeine Operatoren 437
4.2.1.2 Wie man so etwas deklariert 439
4.2.1.3 Und wie man es benutzt 440
4.2.2 Spezielle Operatoren 441
4.2.2.1 Der Zuweisungs-Operator „=“ 441
4.2.2.2 „new“ und „delete“ 443
4.2.2.3 Funktionsaufruf als Operator 450
4.2.2.4 Klassen als Möchtegern-Vektoren 451
4.2.2.5 Intelligente Zeiger: "->" und das unäre ""* 453
4.2.2.6 Inkrement und Dekrement 454
4.2.2.7 Ein- und Ausgabe 456
4.2.3 Beispielprogramm: Die „string“-Bibliothek 457
5. Der Preprozessor 465
5.1 Allgemeines 465
5.2 Was Sie schon immer über #define wissen wollten 466
5.2.1 Makrodefinition 466
5.2.2 Makros mit Parametern 469
5.3 Was Sie noch nie über #include wissen wollten 470

5.4 Bedingte Übersetzung 471
5.5 Pragmatismus 473
5.5.1 Allgemeines und Wiederholungen 473
5.5.2 Breakpoint-Kontrolle 474
5.5.3 Amiga-spezifische Systemaufrufe 475
5.6 Noch mehr Features 476
5.6.1 Verbinden von Zeilen 477
5.6.2 Verbinden konstanter Zeichenketten 477
5.6.3 Trigraph-Sequenzen 477
5.6.4 Hausgemachte Fehlermeldungen 479
5.6.5 Getürkte Dateinamen und Zeilennummern 479
5.6.6 Nix dahinter 480
5.7 Vordefinierte Symbole 480
5.8 Arbeitsweise des Preprozessors 482

6. Was Sie als Amiga-Entwickler wissen sollten 485
6.1 Die Schnittstelle zum Betriebssystem 485
6.2 Start von der Workbench 485
6.2.1 Die Do-It-Yourself-Methode: die Funktion "wbmain" 485
6.2.2 Die einfache Methode: die Datei "wbstartup.h" 488

7. Templates 491
7.1 Einführung 491
7.2 Klassen-Templates 491
7.2.1 Ein einfaches Beispiel 491
7.2.2 Deklarationen und Definitionen 493
7.2.3 Member-Funktionen 495
7.2.4 Statische Member 496
7.2.5 Friends 497
7.3 Funktions-Templates 498
7.3.1 Beispiel: Sortieren von Vektoren 498
7.3.2 Deklaration und Definition 499
7.3.3 Handgestrickt kontra compilergeneriert 499
7.3.4 Argumemt-Matching beim Aufruf von Templatefunktionen 501
7.3.4.1 Benutzung von Funktionstemplates 501
7.3.4.2 Mischung von Funktiontemplates und "echten" Funktionen 502

7.3.5 Generierung von Templatefunktionen 503
7.4 Beispielprogramm: Schon wieder verkettete Listen 504

8. Ausnahmebehandlung 509
8.1 Worum gehts? 509
8.2 Vom Werfen und Auffangen 510
8.3 Von Ausnahmen und ihren Typen 512
8.4 Vom Fangen und wieder Wegwerfen 515
8.5 Von Konstruktoren und Destruktoren 517
8.6 Von erwarteten und unerwarteten Ausnahmen 520
8.7 Beispielprogramm: Das Übliche 523

Referenz 525

1. Die ANSI C Includedateien 527
1.1 Verschiedene nützliche Funktionen: <stdlib.h> 527
1.1.1 Umwandlung zwischen Zahlen und Strings 527
1.1.2 Mathematische Funktionen 529
1.1.3 Zufallszahlen 531
1.1.4 Speicherverwaltung, Typen und Zeiger 532
1.1.5 Programmende 534
1.1.6 Kommunikation mit dem Betriebssystem 535
1.1.7 Sortieren und Suchen 535
1.2 Ein- und Ausgaben: <stdio.h> 536
1.2.1 Typen, Objekte und Konstanten 536
1.2.2 Öffnen und Schließen von Dateien 538
1.2.3 Strings und Zeichenketten 540
1.2.4 Formatierte Ausgabe 542
1.2.5 Formatierte Eingabe 546
1.2.6 Dateioperationen 549
1.2.6.1 Dateiende und Fehlerbehandlung 549
1.2.6.2 Puffern 550
1.2.6.3 Externe Dateien 552
1.2.6.4 Binäre Daten 553
1.2.6.5 Positionieren in Dateien 554
1.2.7 Sonstige Funktionen 555

1.3 Fehlersuche: <assert.h> 556
1.4 Zeichenklassen: <ctype.h> 557
1.5 Fehlernummern: <errno.h> 559
1.6 Ganzzahlige Grenzwerte: <limits.h> 561
1.7 Mathematische Funktionen: <math.h> 562
1.7.1 Umwandlung von Zahlen in Zeichenketten 562
1.7.2 Fließkomma-Berechnungen 564
1.8 Haarsträubende Sprünge: <setjmp.h> 567
1.9 Signale und Ereignisse: <signal.h> 570
1.10 Variable Argumentlisten: <stdarg.h> 572
1.11 Implementationsabhängige Definitionen: <stddef.h> 575
1.12 Zeichenketten und Speicherverwaltung: <string.h> 576
1.13 Datum und Uhrzeit: <time.h> 582

2. Bibliotheken für C++ 587
2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h> 587
2.1.1 Ausgabe 587
2.1.1.1 „ostream" und der Operator „«" 587
2.1.1.2 Basen und Manipulatoren 589
2.1.1.3 Formatierte Ausgabe 591
2.1.1.4 Ausgabe binärer Daten 592
2.1.2 Eingabe 593
2.1.2.1 Texteingaben mit „»" 593
2.1.2.2 Eingabefunktionen auf „istream" 594
2.1.3 Weitere Funktionen 596
2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem 596
2.1.3.2 Fehlererkennung 596
2.2 Ströme und Dateien: <fstream.h> 597
2.2.1 Klassen für Dateien 597
2.2.2 Dateien öffnen und schließen 598
2.2.3 Fehlererkennung 599
2.2.4 Konstruktoren und Destruktoren 600
2.2.5 Puffer 601
2.2.6 Positionieren in Strömen 601

2.3 Ein paar wichtige Definitionen in <new.h> 602
2.4 Ausnahmebehandlung: <exception.h> 605
2.5 Eine String-Library: <tools/str.h> 606
2.5.1 Die Klasse „String" 606
2.5.2 Member-Funktionen 607
2.5.3 Der Operator „+" auf Strings 609
2.5.4 Vergleiche 610
2.5.5 Ein- und Ausgabe 611
2.6 Interna von MaxonC++: <streamdefs.h> 612
2.6 Funktionen für originelle Linker-Opt.: <linkerfunc.h> 615

Easy-Objects 617
Einführung 619
Die Ausnahmebehandlung 623
Container und Cursor 623
Das Handlerkonzept 625
Die Fensterklasse WindowC 627
Die Fensterklasse GTWindowC 630
Gadgets und Gadgetlisten 630
Der Layouter 632
Die Regeln zur Bescheibung eines Layouts 633
Der Layoutvorgang 639
Eine Knopfzeile 639
Eine Gadgetspalte 641
Die Grenzen des Layouters 643
Zur Gestaltung der Oberfläche 643
Die Fensterklasse LayouterWindowC 644
Die Fensterklasse StandardWindowC 645
Datei- und Verzeichnissperren 646
Die Dateiklassen 647

Kommandozeilenargumente 648
Beschreibung des HotHelp Projekts zu EASY-OBJECTS 648
Einführung 648
Hintergrund 648
Technische Kapitel 649
Benutzung des HotHelp Projekts EASY-OBJECTS 651

Source Level Debugger 655

1. Einführung 657
Was können Sie mit MaxonSDB tun? 657
Über dieses Handbuch 657

2. Die Benutzung von MaxonSDB 659
Die Vorbereitung Ihres Programms 659
Programmstart 659
Der Workbenchstart 659
Der Shellstart 660
Der Start aus der Entwicklungsumgebung 661

3. Die Fenster des Debuggers 662
Drag&Drop 662
Das Projektfenster 662
Das Quelltextfenster 663
Breakpoints 664
Das Modulfenster 664
Das Fenster der globalen Variablen 665
Den Knopf Inspiziere. 665
Das Fenster der Funktionen 666
Das Fenster der lokalen Variablen 667
Das Fenster der aktuellen Variablen 667
Das Fenster zur Überwachung 667
Die Konfiguration der Oberfläche 668

Das Menü Projekt 669
Das Menü Schritt 669
Das Menü Fenster 670
Das Menü Einstellungen 670

Anhang 671
Der Enforcer 671
Der SegTracker 671
inline-Funktionen 671
Technische Einzelheiten: 672
Forbid() und Disable() 672
Die Prozessoren 68020,68030 und 68040 673
Sicherheitsabfragen 674
Fehlermeldungen 675

HotHelp 3 677

1. Über diese Anleitung 679

2. Jederzeit hilfsbereit 680

3. Eigenschaften von HotHelp 681

4. Anforderungen 683

5. Installation 683
5.1. Installation auf Festplatte 683
5.2. Installation auf Disketten 684
5.3. Der Installationsvorgang 685
5.4. Installation von Hand 686
5.5. Installation mit englischer Benutzerführung 687

6. Die Bedienung von HotHelp 688
6.1. Die Titelzeile des HotHelp-Fensters 689
6.2. Rollsymbole 690
6.3. Querverweise 690
6.4. Das Vortext-Symbol 691
6.5. Blättern im Projekt 691
6.6. Schlüssel, Projekte und Muster 691
6.7. Abrufen der Projektübersicht 692
6.8. Die Kapitelübersicht 693
6.9. Der Export von Hilfstexten 693
6.10. Verschiedene Ansichten über HotHelp 694
6.11. Die integrierte Suchfunktion 695
6.12. Verwendung von Lesezeichen 696
6.13. Hilfe über Hilfe 696
6.14. Tastaturbedienung 697
6.15. Möglichkeiten, HotHelp aufzurufen 697

7. Onlineprojekte 699
7.1. Benutzung der Onlineprojekte 699
7.2. Anwendungsgebiete für Onlineprojekte 700
7.3. Onlineprojekte und andere Editoren 700

8. ARexx 701

9. Änderungen seit Version 2 702

10. Die HotHelpTools 704
10.1. Die Hilfsfunktion der HotHelpTools 704
10.2. Start der HotHelpTools 705
10.3. HotHelpPref - HotHelps Voreinstellungen 706
10.3.1. Fenster 1: Farben 707

10.3.2. Fenster 2: Zeichensatz 708
10.3.3. Fenster 3: Ausgabeformate 709
10.3.4. Fenster 4: Steuerung 710
10.3.5. Fenster 5: Hotkeys 711
10.3.6. Fenster 6: Online-Optionen 711
10.3.7. Fenster 7: HKH-Optionen 712
10.4. HotHelpPro - Der Projektmanager 714
10.4.1. Standardprojekte 714
10.4.2. Onlineprojekte 715
10.4.3. Projekttest 716
10.4.4. AmigaGuide-Projekte 716
10.5. HotHelpMarker - der Projekte-Editor 718
10.5.1. Grundlagen der Projekterstellung 718
10.5.2. Hintergrund 719
10.5.3. Eingabe der Texte 719
Die Quelldateien 720
Die Texte 720
10.5.4. Markieren der Texte 720
Der HotHelp-Textmarker 720
Aufteilung des Textes 721
Einteilung in Einträge 721
Die Schlüssel 722
Querverweise 722
Grafiken 723
Schriftarten 723
Die Übersetzung 723
10.5.5. Resümee 724
10.5.6. Weitere Gestaltungsmöglichkeiten 724
Titel 725
Kapitel, Vorgänger und Nachfolger 725
Projekt- und ARexx-Querverweise 725
Schlüsselgruppen 725
10.6. EasyHotHelp 727
10.7. HotHelpComp - der HotHelp-Übersetzer 728
11. Letzte Worte 729

MaxonASM 731

1. Einleitung 733

2. Grundlagen der Assemblerprogrammierung 734

2.1 Warum Assembler? 734

2.2 Der Aufbau eines Quelltextes 734

3. Der Assembler 735

3.1 Die Adressierungsarten des MC 68000 735

3.2 Symbole und mathematische Ausdrücke 737

3.2.1 Globale Symbole 737
3.2.2 Lokale Symbole 738
3.2.3 Label 738
3.2.4 Variablen 738
3.2.5 Reservierte Symbole 739
3.2.5.1 NARG 739
3.2.5.2 __RS 739
3.2.5.3 _MOVEMBYTES 739
3.2.6 Mathematische Ausdrücke 740
3.2.7 Zahlenformate 740
3.2.8 Operationen und deren Prioritäten 740

3.3 Macros 741

Schlüsselwörter/Funktionen 743
1. *VALOF(<Symbol>) 743
2. *STRLEN(<String>) 744
3. *LEFT(<String>,n) 744
4. *RIGHT(<String>,n) 744
5. *MID(<String>,s,n) 744
6. *UPPER(<String>) 745

3.4 Bedingte Assemblierung 746

3.5 ALIGNment 747

3.6 Sektionierung 748

Direktiven zur Steuerung der Code-Erzeugung 749

3.7 Erzeugung von absolutem Code 750

3.8 Konstanten 751
3.9 Reservierte Speicherblöcke 753
3.10 Definition einer Variablen 753
3.11 Definition von Strukturoffsets 754
3.12 Externe Quelltexte 754
3.13 Externe Rohdateien 755
3.14 Copper-Assembler 755
3.15 Externe Symbole 758
3.16 Codearten 760
3.17 Assemblerbetriebsarten 760
3.18 Mögliche Optimierungen 761
3.19 Ausgabedateien 763

Anhang A 765
Die Befehle des MC 68000 / MC 68010 765
Die zusätzlichen Befehle des MC 68010 768

Anhang B 769
Die Assemblerdirektiven 769

Anhang C: Die Fehlermeldungen 773
DOS-Fehler beim Dateihandling 773
Fehler in der Sektionierung 773
MACRO-Fehler 773
Fehler beim Parsen einer Zeile 775
Fehler bei der Labeldefinition 775
Fehler im math. Ausdruck 775
Fehler im Addressmode 776
Fehler im ersten Befehlsoperanden 777
Fehler im zweiten Befehls-Operanden 778

Fehler im dritten Befehls-Operanden 778
Fehler in den IF-Konstruktionen 778
Spezielle Fehler bei Branch-Befehlen 779
Spezielle Fehler bei EQU 779
Spezielle Fehler bei EQUR 779
Spezielle Fehler bei REG 779
Spezielle Fehler bei SET 780
Spezielle Fehler bei CNOP/ALIGN 780
Spezielle Fehler bei XDEF 780
Spezielle Fehler bei XREF 780
Spezielle Fehler bei DC.x 780
Spezielle Fehler bei RS.x 780
Spezielle Fehler bei INCBIN 781
Spezielle Fehler bei BASEREG 781
Allgemeine Fehler 781
Interne Assembler-Fehler 781
Fehler bei FPU-Operationen 782
Fehler im Addressmode 782
Fehler im zweiten Befehlsoperanden 782
Fehler im Addressmode 782
Fehler im ersten Operanden 783
Fehler im zweiten Operanden 783
Fehler bei Copper-Operationen 783

Anhang D: Register 784

Anhang E, Befehle 785

AMIGA

MaxonC++

Anwenderhandbuch

MAXON computer

Maxon C++ 3

1.1 Was gibt es Neues?

Das MaxonC++ Compiler-System hat eine ganze Menge neuer Features bekommen. Das wichtigste ist sicher die Kompatiblität zum 3.0-Standard, weshalb auch die Versionsnummer 2.0 übergangen wurde. Der "3.0-Standard" wird wohl weitgehend mit dem lange erwarteten und immer noch nicht ganz fertigen ANSI C++-Standard identisch sein. Die Neuerungen gegenüber dem "2.0-Standard" (dem auch Maxon C++ 1.x folgte) sind zum einen die "Templates", durch die das Konzept der abstrakten Datentypen besser realisiert werden kann, und zum anderen die Ausnahmebehandlung, die die Zuverlässigkeit von Programmen zu steigern hilft.

Neben vielen anderen Verbesserungen bietet der MaxonC++ Compiler jetzt auch die Möglichkeit, speziellen Code für 32Bit-Prozessoren und FPUs erzeugen zu lassen. Schon seit der Version 1.1 ist es möglich, Includedateien "vorcompilieren" zu lassen. Das beschränkte sich aber auf ein Umkopieren der Dateien in Token-Form, was keine besonders dramatische Beschleunigung des Übersetzungsvorgangs brachte. Jetzt kann der Compiler, nachdem er die Header-Dateien gelesen hat, seinen gesamten internen Zustand abspeichern und beim nächsten Übersetzungsvorgang genau dort wieder einsetzen. Dadurch schrumpfen die Compile-Zeiten für Programme, die viele Includes benutzen, drastisch zusammen.

Die integrierte Umgebung MCPP wurde komplett überarbeitet. Einmal davon abgesehen, daß alles viel schöner und bequemer wurde, ist das erheblich verbesserte integrierte „Make" besonders hervorzuheben. Viele dieser Verbesserungen kamen auch dem „Make" im MCPPC zugute. Der Maxon Assembler (nur in der Developer-Version enthalten) wurde besser in die Umgebung integriert, indem MCPP die Fehlermeldungs-Dateien des Assemblers liest und im Compiler-Fehlermeldungswindow anzeigt. Außerdem besitzt MCPP jetzt einen eigenen ARexx-Port.

Auch beim "Zubehör" hat sich einiges getan: Es gibt einen komplett neugeschriebenen und in vieler Hinsicht verbesserten Debugger sowie eine Klassenbibliothek, die nicht nur eine objektorientierte Schnittstelle zu den wichtigsten Bereichen des Amiga-Betriebssystems darstellt, sondern viele alltägliche Programmieraufgaben erheblich vereinfacht.

Die einzige Änderung, die zu Inkompatiblitäten mit der ersten Version führen kann, ist die Umstellung des Datentyps "float" vom "FFP-Single"- auf das "IEEE-Single"-Format. Dadurch erhält "float" die Genauigkeit und den Wertebereich, den der ANSI-C-Standard fordert.

1.2 Der Umstieg

1.2.1 Konfigurationsdateien

Ihre alte Konfigurationsdatei „*S:MaxonC++.config*" können Sie leider nicht mehr verwenden.Wenn Sie den alten Compiler nicht mehr verwenden, können Sie diese Datei löschen. Die neue Konfiguration heißt „*S:mcpp.config*" und hat ein vollständig anderes Format. Dafür gibt es jetzt die Möglichkeit, alle Einstellungen einschließlich Größe und Lage der verschiedenen Fenster in einer Datei „*mcpp.SESSION*" abzuspeichern. Das funktioniert also ganz analog zum „*EDWARD.SESSION*" des Editors.

1.2.2 „Edward.DEF"

Wenn Sie bereits eine ihren speziellen Bedürfnissen angepaßte Editor-Konfigurationsdatei "*Edward.DEF*" besitzen, werden Sie diese wohl weiter verwenden wollen. Das ist ohne weiteres möglich, allerdings sind beim *MCPP* viele Features neu hinzugekommen und dafür überflüssige entfernt worden, so daß Sie sinnvollerweise Ihre Menü- und Tastenbelegung überarbeiten sollten. Dabei werden Sie zwei neue Edward-Features nützlich finden: Zum einen kann in der "*Edward.DEF*"-Datei mit Steuerzeilen wie `"#ifdef MCPP1"`, `"#ifdef MCPP3"` oder `"#ifdef MPASCAL3"` zwischen den verschiedenen Applikationen, in die Edward inzwischen eingebunden wird, unterschieden werden (bisher gab es ja nur die allgemeine Modusabfrage mit `"#ifdef COMPILER"`); zum anderen können in "*Edward.DEF*" jetzt Include-Dateien verwendet werden. Da die Edward-Konfiguration so auf verschiedene Dateien verteilt wird, ist sie bei künftigen Updates leichter zu pflegen.

Wichtig

Bei der Installation des neuen Compilers wird eine eventuell von einer alten Version von EDWARD vorhandene „EDWARD.DEF"-Datei umbenannt und die neue installiert.

Sollten Sie modifikationen an Ihrer alten „EDWARD.DEF"-Datei vorgenommen haben, müssen Sie diese in die neue übertragen.

Die Abkürzungen für MCPP-Kommandos werden wegen Zweifels an deren Nützlichkeit generell nicht mehr unterstützt. Es muß jetzt also z. B. `SendAppComm "compile"` statt `SendAppComm "c"` heißen.

Alle alten und neuen Funktionen der Projektverwaltung sind jetzt über einen ganz neuen Dialog erreichbar. Dadurch kann das überladene, unübersichtliche und umständliche **„Make"-Menü** komplett entfallen. Auch sonst wurden einige Anweisungen überflüssige oder zumindest wenig nützliche Anweisungen weggelassen. Die Kommandos:

`addfiletx, addobj, mkinstr, mkdep, remobj, remtx, touch, opentext, chdir und compfile`

werden nicht mehr unterstützt, während die Make-Anweisungen

```
newmk, loadmk, addedittx, addtxobj und leavemk
```

prinzipiell weiter benutzt werden können. Am besten ist es aber, das **„Make"-Menü** durch einen schlichten Menüpunkt wie

```
ITEM "Projektverwaltung" (SendAppComm "make") "M"
```

im **„Compiler"-Menü** zu ersetzen. Diese Funktion öffnet den erwähnten Make-Requester - oder den Filerequester, falls noch keine Projektdatei geladen ist. Im letzteren Fall können Sie sowohl eine bestehende Projektdatei auswählen als auch den Namen einer neu zu erzeugenden Datei angeben.

Eine andere sinnvolle Ergänzung wäre ein Menüpunkt

```
ITEM "Fehlermeldungen" (SendAppComm "errors")
```

(natürlich können sie genausogut auch eine Taste mit dieser Funktion belegen).

Damit ist es jetzt möglich, das bereits geschlossene Compiler-Window erneut zu öffnen, z. B. um einen erneuten Blick auf die letzten Fehlermeldungen zu werfen. `SendAppComm "saveconf"` öffnet in der aktuellen Version keinen Filerequester mehr, sondern schreibt die Konfiguration grundsätzlich nach *„S:mcpp.config"*. Übrigens enthält diese Datei jetzt endlich wirklich **alles**, was es über die Umgebung und ihre Einstellungen zu sagen gibt. Neu ist `SendAppComm "savesession"` Diese Anweisung schreibt die Konfiguration in eine Datei *„mcpp.SESSION"* im aktuellen Verzeichnis, von wo sie beim nächsten Programmstart aus dem gleichen Verzeichnis automatisch geladen wird. Und weil dies analog zur Session-Datei des Edward ist, sollte überall, wo in *„Edward.DEF"* das *„SaveSession"* des Editors aufgerufen wird, das *MCPP-"SaveSession"* ergänzt werden. Aus einem Menüpunkt

```
ITEM "Projekt speichern"  (SaveSession)
```

könnten Sie also

```
ITEM "Projekt speichern" ((SaveSession)(SendAppComm "savesession"))
```

machen.

Eine Compiler-Menüdefinition könnte z. B. so aussehen:

```
TITLE "Project"  ... Normale Editor-Projekt-Menüs ...
ITEM "Save Session"  ((SaveSession) #ifdef COMPILER (SendAppComm
"savesession")#endif ) "U"
#ifdef MCPP3
TITLE "Compiler"
ITEM "übersetzen"   (Sendappcomm "compile")
ITEM "Debugger"     (SendAppComm "debug") "D"
EMPTY
ITEM "Ausführen"    (SendAppComm "run")
ITEM "Abbrechen"    (SendAppComm "break")
```

```
EMPTY
ITEM "Exe-Datei"    (SendAppComm "exefile")
ITEM "Objektdatei"  (SendAppComm "objectfile")
EMPTY
ITEM "Information"  (SendAppComm "info")
ITEM "Fehlermeldungen"  (SendAppComm "errors")
TITLE "Make"
ITEM "Neues Makefile"  (SendAppComm "newmk")
ITEM "Makefile laden"  (SendAppComm "loadmk")
ITEM "Projektverwaltung"  (SendAppComm "make") "M"
EMPTY
ITEM "Text aus Editor aufnehmen"  (SendAppComm "addedittx")
ITEM "Text und Objekt aufnehmen"  (SendAppComm "addtxobj")
EMPTY
ITEM "Make verlassen"  (SendAppComm "leavemk")
TITLE "Optionen"
ITEM "System"  (SendAppComm "opsys")
ITEM "Compiler"  (SendAppComm "opcomp")
ITEM "Linker"  (SendAppComm "oplink")
EMPTY
ITEM "Compiler-Optionen sichern"  (SendAppComm "saveconf")
ITEM "Compiler-Session sichern"(SendAppComm "savesession")  #endif
```

Das folgende wäre eine praktische Tastenbelegung:

```
#ifdef COMPILER
(
F9              (Sendappcomm "Run")
F10             (Sendappcomm "ExeFile")   )

(SHIFT
F9              (Sendappcomm "Compile")
F10             (Sendappcomm "ObjectFile")   )

(CTRL
F9              (Sendappcomm "Errors")
F10             (Sendappcomm "Info")   )
#endif
```

1.2.3 Dateien und Verzeichnisse

Alle ausführbare Programme liegen jetzt im Verzeichnis „*MCPP:bin*“, was in früheren Versionen nicht immer so einheitlich geregelt war. Genau dort sollte auch die Datei „*MCPP.RCT*“ liegen. Sie ist nötig, weil die neue Oberfläche von MCPP mit MakeAPP erzeugt wurde.

1.3 Ein ersterÜberblick

1.3.1 Disketten

MaxonC++ wird auf mehreren Disketten ohne Kopierschutz geliefert. Die genaue Zahl und den Inhalt der Disketten erfahren Sie, wenn Sie die *„ReadMe"-Datei* auf der ersten Diskette lesen. Da sich nach Drucklegung dieses Handbuchs noch Änderungen ergeben können, wählten wir diesen Weg.

1.3.2 Installation

Um das System auf Ihrer Festplatte zu installieren, sollten Sie sich zuerst ein geeignetes Verzeichnis anlegegen, z. B. *„DH0:MCPP"*. Sie sollten Dateinamen mit einem „+" am Ende meiden, da das CLI bzw. die Shell bei Eingaben, die mit „+" enden, Schwierigkeiten bereiten.

Nun starten Sie einfach das Installationsprogramm von der ersten Diskette.

1.3.3 Die Komponenten des MaxonC++ Systems

In diesem Abschnitt sollen die wichtigsten Dateien des MaxonC++ Systems kurz erläutert werden. Das soll Ihnen helfen, sich mit den zahlreichen Dateien und Verzeichnissen besser zurechtzufinden.

Bei der Installation wird ein logisches Gerät *„MCPP:"* dem Verzeichnis, in das Sie die Installation durchgeführt haben, zugewiesen. Darin befindet sich ein Unterverzeichnis *„Bin"*, in dem die meisten ausführbaren Dateien stehen.

"MCPP" ist die integrierte Entwicklungsumgebung. Wenn die Installation korrekt und erfolgreich abgelaufen ist, können Sie versuchen, dieses Programm zu starten.

MCPP enthält das komplette Entwicklungssystem einschließlich Compiler und Projektverwaltung - nur der Editor muß noch dazugeladen werden. Während der Installation wird der Editor Edward in ein Verzeichnis *„Edward:"* abgelegt, von wo MCPP ihn automatisch nachlädt. Sie können Edward aber auch allein ohne Compiler aufrufen.

Edward benötigt unbedingt eine Datei *„Edward:Edward.DEF"*, in der die Menü- und Tastenbelegung der Entwicklungsumgebung abgespeichert ist. Diese Datei wird bei der Installation autmatisch erzeugt. Im Edward-Handbuch finden Sie eine ausführliche Anleitung, wie Sie diese Datei ändern können, um das MaxonC++ nach Ihren persönlichen Vorstellungen und Wünschen zu konfigurieren.

Sowohl MCPP als auch Edward benötigen die *„rct.library"*. Diese sollte während der Installation in Ihr *„Libs:"*-Verzeichnis kopiert worden sein.

Neben einigen nützlichen anderen Funktionen erleichtert die *„rct.library"* insbesondere die Programmierung von Benutzeroberflächen, die mit dem Oberflächengenerator *„MakeAPP"* erzeugt wurden. Ein solches Programm benötigt dann aber nicht nur diese Library, sondern auch eine

„*.RCT*"-*Datei*, in der die eigentliche Oberfläche abgespeichert ist. Für MCPP heißt diese Datei „*MCPP.RCT*" und wird im Verzeichnis „*MCPP:Bin*" erwartet, während die Edward-Oberfläche in „*EdwardRes.RCT*" aus dem Verzeichnis „*Edward:*" abgespeichert ist.

Um MCPP zu starten, müssen also zusammengefaßt folgende Dateien installiert sein und im richtigen Verzeichnis liegen:

- *MCPP:Bin/MCPP*
- *MCPP:bin/MCPP.RCT*
- *EDWARD:Edward*
- *EDWARD:Edward.DEF*
- *EDWARD:MCPP3.DEF*
- *EDWARD:EdwardRes.RCT*
- *LIBS:rct.library*

In der Entwicklungsumgebung können Sie über Dialoge eine ganze Menge Einstellungen vornehmen. Damit Sie das nicht bei jedem Start des Systems neu tun müssen, können Sie die Einstellungen natürlich auch abspeichern, und zwar in zwei Dateien im Verzeichnis „*S:*":

- „*S:Edward.CONF*" für Editor-Einstellungen,
- „*S:MCPP.config*" für Compiler-Optionen.

Auch diese beiden Dateien werden bei der Installation automatisch angelegt. Wenn Sie fehlen, macht das aber nichts, da sowohl MCPP als auch Edward sinnvolle Default-Werte besitzen. Vorzugsweise hält das MaxonC++-System sich ohnehin nicht an die Dateien im „*S:*"-Verzeichnis, sondern versucht nach Möglichkeit, aus dem aktuellen Verzeichnis folgende Dateien zu lesen:

- „*Edward.SESSION*" für Editor-Einstellungen,
- „*mcpp.SESSION*" für Compiler-Optionen.

Diese beiden Dateien enthalten nicht nur sämtliche Einstellungen und Optionen, sondern auch Informationen darüber, welche Texte geladen und welche Fenster wo geöffnet sind. Die Session-Dateien werden aber nicht bei der Installation erzeugt, sondern auf Verlangen von *MCPP* bzw. Edward selbst ins aktuelle Verzeichnis geschrieben. Wenn eine dieser Dateien vorhanden ist, hat sie Vorrang vor der jeweils entsprechenden Konfigurationsdatei im Verzeichnis "S:". Durch die Session-Dateien können Sie den aktuellen Zustand am Ende einer "Sitzung" abspeichern und später wieder genau so wieder anfangen, wie Sie aufgehört haben.

"*mcppc*" ist die Kommandozeilenversion des Compilers. Falls Sie integrierte Entwicklungsumgebungen nicht mögen sollten, werden Sie wahrscheinlich dieses Programm von einer Shell aus benutzen. Naheliegenderweise liegt „*mcppc*" im Verzeichnis „*MCPP:Bin*".

Außerdem finden Sie im „*MCPP:bin*"-Verzeichnis (mindestens) folgende Programme:

- *MaxonASM:* Der Assembler (nur in Developer-Version)
- *MaxonSDB:* Der Source-Level-Debugger (nur in Developer-Version)

Ferner sollte nach erfolgreicher Installation auch irgendwo ein Verzeichnis namens *„HOTHELP:"* existieren, in dem alle Dateien und Verzeichnisse liegen, die zum Online-Hilfesystem *„HotHelp"* gehören.Um sinnvoll mit MaxonC++ arbeiten zu können, benötigen Sie auch Libraries und Include-Dateien. Die Standard-Bibliotheken befinden sich normalerweise im Verzeichnis *„MCPP:Lib"*, während in *„MCPP:Include"* sowohl die Standard-Includes von MaxonC++ als auch die jeweils aktuellsten Amiga-OS-Includes von Commodore liegen. Wenn Sie diese Dateinamen ändern wollen, sollten Sie wissen, was sie tun, und dieses Benutzerhandbuch aufmerksam gelesen haben.

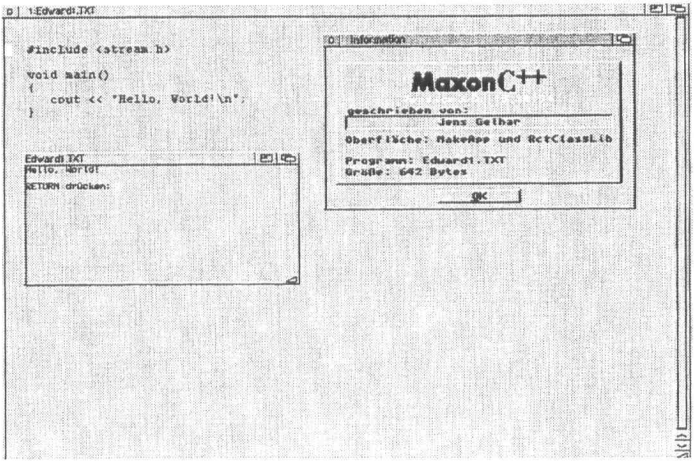
1.4 Erste Schritte mit MCPP

Die integrierte Entwicklungsumgebung *"MCPP"* kann wahlweise von der Workbench oder vom CLI gestartet werden. Dabei stellt *"MCPP"* noch keine vollständige Benutzeroberfläche dar, sondern lädt noch den Editor *"Edward"* nach. Folglich muß dieser Editor korrekt installiert sein. Wird der Edward nicht gefunden, steigt *"MCPP"* gleich mit einem entsprechenden Requester aus.

Nach dem Start landen Sie also automatisch im Editor. Hier können Sie Ihre C++-Programme schreiben und bearbeiten. Dem Edward ist ein separates Handbuch gewidmet, deshalb soll hier nicht näher auf die Editor-Funktionen eingegangen werden.Am besten machen Sie sich jetzt ein wenig mit dem Editor vertraut, indem Sie ein kleines Programm schreiben. Sie könnten z. B. folgendes unvermeidliche und traditionsreiche Prograämmchen abtippen:

```
#include <stream.h>
void main()
{ cout << "Hello, World!\n"; }
```

Nun haben Sie ein Programm als Quelltext vorliegen und können den Compiler ausprobieren. Zuerst sollten Sie den Text abspeichern, um ihm so einen sinnvollen Namen zu geben, mit dem der Compiler auch etwas anfangen kann. Dann wählen Sie einfach den Menüpunkt **„übersetzen"** aus der Menüleiste und warten ab, was geschieht.



Es öffnet sich ein Fenster, in dem *MCPP* einige mehr oder weniger kryptische Meldungen ausgibt. Tritt während der Übersetzung ein Fehler auf, wird er in diesem Fenster gemeldet und der entsprechende Quelltextbereich angezeigt. Sie können dann eine Fehlermeldung anwählen und mit den Gadgets ***„Anspringen"*** und ***„In Editor"*** direkt die entsprechende Textstelle im Editor anspringen. Wenn das System korrekt installiert wurde und ihr Programm ebenfalls fehlerlos ist, sollte das Fenster aber ebenso schnell verschwinden, wie es aufgetaucht ist. Nun können Sie die Menü-Funktion **„Ausführen"** anwählen und das Programm starten. Auch dabei öffnet sich ein Fenster, in dem (wer hätte das gedacht!) **"Hello, World!"** ausgegeben wird. Außerdem werden Sie freundlich aufgefordert, die <Return>-Taste zu drücken, denn sinnvollerweise schließt *MCPP* das Window nicht, bevor Sie Gelegenheit hatten, die Ausgabe zu lesen.

Sie können das Programm natürlich noch einmal starten. Falls Sie aber seit der letzten Übersetzung etwas am Quelltext verändert haben, wird automatisch der Compiler erneut gestartet.

2. Die integrierte Entwickungsumgebung

2.1 Programmstart und Optionen

Wenn Sie *MCPP* vom CLI bzw. aus einer Shell starten, können Sie optional einige Parameter angeben. Als erstes können Sie einen Dateinamen angeben, z. B.

```
run MCPP MCPP:Demos/hello.c
```

Sofern die angegebene Datei existiert und lesbar ist, wird sie in den Editor geladen. Da der Edward mehrere Texte verwalten kann, können Sie natürlich auch mehrere Dateinamen angeben.

Wenn Sie eine andere Konfigurationsdatei als *„s:MaxonC.config"* benutzen wollen, können sie ein zusätzliches Argument mit vorangestelltem `"-c"` angeben. Ein Beispiel:

```
run MCPP test.c -c MCPP:spezial.config
```

Im Gegensatz zur ersten Version von *MCPP* darf zwischen `"-c"` und dem Dateinamen jetzt auch ein Leerzeichen stehen. Da man sich spezielle Einstellungen jetzt aber auch in der Datei *"mcpp.SESSION"* abspeicher kann, gibt es nur noch wenig Gründe, dieses Feature zu benutzen.

MaxonC^{++} besitzt eine integrierte Projektverwaltung, allgemein bekannt als *„Make"*. Sie können gleich beim Programmstart eine solche Make-Datei öffnen, und zwar mit der Option `"-m"`:

```
run MCPP -m plot.makefile.
```

Natürlich lassen sich diese Optionen und Argumente in beliebiger Reihenfolge kombinieren, z. B.

```
MCPP fclass.c -m wagga.makefile -c df1:s/MaxonC.config.
```

Seit Version 3 besitzt *MCPP* einen eigenen ARexx-Port. Wenn Sie als einziges Argument `"-r"` angeben, also z. B.

```
run MCPP -r
```

starten MaxonC^{++} im ARexx-Modus. Dabei wird Edward nicht geladen. Statt dessen wird ein ARexx-Port namens `"MCPP"` eingerichtet, mit dem das System über ein ARexx-Script gesteuert werden kann. Auch für das Nachladen des Editors und die Kommunikation zwischen den beiden Programmen müssen Sie dann (über geeignete ARexx-Programme) selbst sorgen. So können Sie einen beliebigen - oder auch überhaupt keinen - Editor in die Umgebung einbinden.

Hinter der Option `"-r"` kann optional noch ein Portname angegeben werden, z. B,

```
run MCPP -r Wagga
```

In diesem Fall wird der ARexx-Port nicht `"MCPP"`, sondern `"WAGGA"` genannt. Man beachte, daß der Portname in Großbuchstaben umgewandelt wird. In jedem Fall aber gilt, daß MaxonC^{++} für eindeutige Portnamen sorgt. Falls z. B. schon ein Port `"MCPP"` existiert, werden so lange `"MCPP1"`,

`"MCPP2"`, `"MCPP3"` usw. ausprobiert, bis ein Name gefunden wird, den es im ganzen System noch nicht gibt.

Alles weitere über den ARexx-Port und was Sie damit anfangen können entnehmen Sie bitte Abschnitt 2.6.

2.2 Compiler und Editor - eine harmonische Beziehungskiste

Wie oben schon angedeutet, ist der Editor *Edward* ein eigenständiges Programm, das auch ohne den Compiler benutzt werden kann. *MCPP* lädt ihn einfach nach und startet ihn als zweiten Prozess, läßt aber keinen Zweifel daran aufkommen, daß er der Boß ist.

Weniger blumig ausgedrückt, richtet *MCPP* einen Port ein, über den sich Editor und Compiler Nachrichten zuschicken. Wird im Edward vom Benutzer eine Compiler-Funktion aufgerufen, etwa durch eines der Pull-Down-Menüs oder eine entsprechend belegte Tastenkombination, läßt Edward dem Compiler eine entsprechende Nachricht zukommen.

Ich erzähle ihnen diese ganzen implementatorischen Details, damit Sie verstehen können, wie Sie sich das MaxonC++-System selbst konfigurieren können. Der Schlüssel dazu ist die „*.DEF*"-Datei des Edward. Hier werden alle Menüs und Tastenbelegungen des Editors abgelegt, und hier muß irgendwie die Benutzerschnittstelle des Compilers integriert werden, auch wenn der Editor nicht die geringste Ahnung hat, was ein Compiler ist und wozu er gut ist.

Deshalb hat der Edward eine Spezial-Funktion namens `"SendAppComm"`, die als Argument wiederum eine Compiler-Funktion erwartet. Beispielsweise gibt es eine *MCPP*-Anweisung namens `"compile"`. Wird nun vom Benutzer, also Ihnen, im Edward ein Menü ausgewählt, das mit der Funktion `SendAppComm "compile"` belegt ist, schickt Edward das `"compile"` als Nachricht an den *MCPP* und kümmert sich dann nicht weiter drum. *MCPP* dagegen erhält diese Anweisung, fordert vom Edward noch ein paar zusätzliche Daten an und beginnt dann freudig mit der Übersetzung. Natürlich kann man nicht nur ein Menü, sondern auch eine Taste oder Tastenkombination mit diesen `"SendAppComm"-Anweisungen` belegen.

Auf diese Weise sind der Compiler und der Editor vollkommen voneinander abgekoppelt, und *Edward* braucht nichts über *MCPP* zu wissen. Nun kommt es natürlich vor, daß Sie den Edward einmal „einfach so" ohne Compiler benutzen wollen. Deshalb läßt Edward einen kleinen „Preprozessor" auf die „*.DEF*"-Datei los und filtert je nach Modus Teile aus dem Text aus - ähnlich wie ein C-Preprozessor bei `"#ifdef"`-Zeilen. Auch dieses Feature ist in ihrem Edward-Handbuch beschrieben. *MCPP* teilt dem *Edward* mit, daß ein Symbol namens `"MCPP3"` definiert sein soll. Nun können Sie Dinge wie `"#ifdef MCPP3"` in Ihr *DEF-File* schreiben, um Definitionen je nach Modus ein- oder auszublenden. Insbesondere werden Sie sich um sämtliche Menü-Definitionen des Compilers eine `"#ifdef MCPP3 ... #endif"`-Klammer legen wollen, wie es auch in der mitgelieferten Standard-Datei der Fall ist.

Langer Rede schwacher Sinn: Wenn im folgenden von irgendwelchen Menüs die Rede ist, werden dabei jedesmal Hinweise wie „...oder wie auch immer Sie diese Funktion konfiguriert haben" unter-

schlage. Gemeint ist damit jeweils immer die Funktion, die in der Standard-Konfiguration auf dem entsprechenden Menü liegt.

Es gibt übrigens standardmäßig folgende Menüs, mit den nachstehend angegebenen Compiler-Funktionsnamen:

Compiler

- Übersetzen	compile
- Compile Datei	compfile
- Ausführen	run
- Abbrechen	break
- Debugger	debug
- Exe-Datei	exefile
- Objektdatei	objectfile
- Information	info
- Fehlermeldungen	errors

Make

- Projektverwaltung	make
- Neues Makefile	newmk
- Makefile laden	loadmk
- Text aus Editor aufnehmen	addedittx
- Text und Objekt aufnehmen	addtxobj
- Make verlassen	leavemk

Optionen

- System	opsys
- Compiler	opcomp
- Linker	oplink
- Optionen sichern	saveconf

2.3 Das Konzept

Auch auf die Gefahr hin, daß ich Sie damit langweilen könnte, möchte ich trotzdem noch ein paar grundsätzliche Anmerkungen loswerden. Prinzipiell kann MCPP in zwei verschiedenen Modi arbeiten: Im **„normalen"**-Modus, in dem man vor allen kleinere Programme (mit nur einem Modul) schreibt, und im **„Make"**-Modus. Da der Projekverwaltung „Make" ein eigenes Kapitel gewidnet ist, befassen wir uns zunächst nur mit dem Standard-Modus, der natürlich auch aktiv ist, wenn Sie das MaxonC^{++}-System starten.

Auch wenn *Edward* beliebig viele Texte gleichzeitig bearbeiten kann, verwaltet *MCPP* immer nur ein Programm. Das hat zum einen Gründe, die mit dem Speicherplatz zusammenhängen, und resultiert

zum anderen aus der Beobachtung, daß das in der Praxis nicht besonders relevant ist, weil man normalerweise zwar durchaus mit mehreren Texten, aber an nur einem Programm arbeitet.

Die Funktion `"compile"` übersetzt stets den Text im gerade aktiven Fenster. Bei `"run"` wird immer zuerst folgendes geprüft:

- **Liegt überhaupt ein übersetztes Programm vor?**
- **Befindet sich der Quelltext dieses Programms im aktiven Editor-Fenster?**
- **Blieb er seit der letzten Übersetzung unverändert?**

Falls eine der drei Fragen mit „Nein" beantwortet werden muß, wird die `"compile"-Funktion` aufgerufen und damit der Text im aktiven Fenster übersetzt.

So, das sollte gesagt werden, damit Sie beim Rumprobieren mit MaxonC++ (was Sie wahrscheinlich schon längst getan haben) nicht in vermeidbare Verwirrung stürzen (was dabei hoffentlich nicht geschehen ist). Und nun folgt endlich eine ausführliche Beschreibung der Funktionen von MaxonC++:

2.4 Compiler-Funktionen

2.4.1 Programme übersetzen

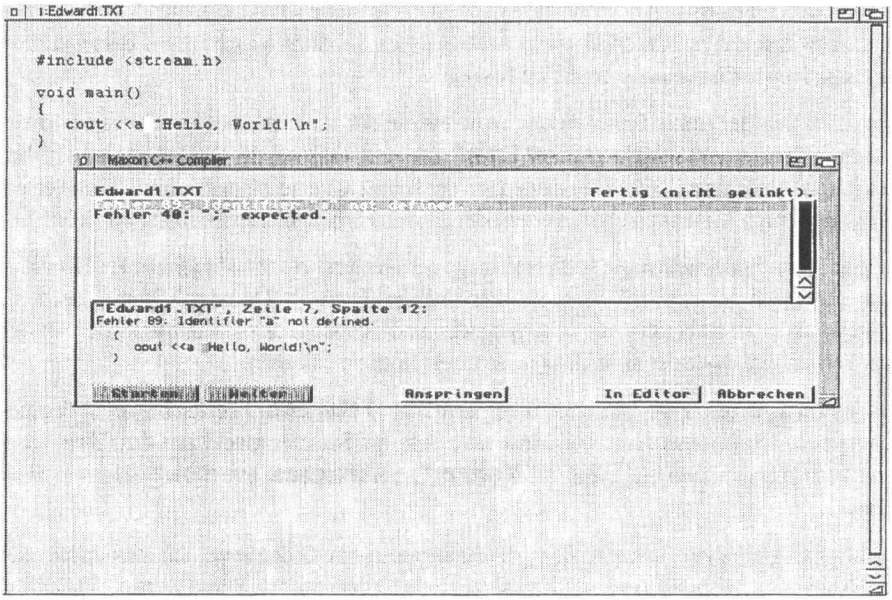
2.4.1.1 Das Compiler-Fenster

Am besten fangen wir mit der Funktion an, wegen der Sie MaxonC++ überhaupt gekauft haben, nämlich dem Übersetzen von Programmen.

Wenn Sie diese Funktion auswählen, wird - je nach Modus - entweder das Programm im gerade aktiven Editor-Fenster übersetzt, oder das integrierte „Make" entscheidet selbst, welche Objektdateien neu übersetzt werden sollen. In jedem Fall öffnet sich aber ein Fenster, das in fünf Bereiche aufgeteilt ist.

Am oberen Rand befindet sich die Statuszeile. Hier wird stets links der Name der gerade übersetzten Datei angezeigt, während *MCPP* rechts sagt, was gerade getan wird bzw. zuletzt getan wurde.

Am besten probieren Sie es einmal aus: Laden Sie einen Quelltext (z. B. eines der Beispielprogramme) in den Editor und wählen Sie den Menüpunkt **"Übersetzen"** aus. Wenn alles klappt, wird oben rechts in schneller Folge (bei kurzen Programmen wohl zu schnell zum Mitlesen) das folgende ausgegeben:



- **"Übersetze":**
  Der Compiler liest und analysiert das Programm.

- **"Keine Fehler gefunden" bzw. "X Fehler gefunden":**
  Nachdem der Quelltext gelesen und auf seine Korrektheit geprüft wurde, erzeugt der Compiler nun den Code dazu.

- **"Programm wird gelinkt":**
  Der Linker bindet das erzeugte Programm mit den nötigen Bibliotheksfunktionen zusammen.

- **"Fertig":**
  Das Programm ist übersetzt und ausführbar. Wurden keine Fehler oder Warnungen ausgegeben, verschwindet das Window bei dieser Meldung automatisch.

Natürlich kann dabei beliebig viel schief gehen, beispielsweise könnte das Programm fehlerhaft sein. Fehlermeldungen und Warnungen aller Art werden in dem Rahmen unterhalb der Statuszeile ausgegeben. *MCPP* gibt immer höchstens so viele Fehler aus, wie in diesen Rahmen passen (das sind zehn Stück) und wartet auf eine Reaktion des Benutzers, falls die Höchstzahl erreicht wurde.

Eine solche einzeilige Fehlermeldung allein ist meist natürlich nicht aussagekräftig genug. Deshalb kann sich der Programmierer jeweils zu einer Meldung ausführlichere Informationen anzeigen lassen.

Unterhalb des Fehlermeldungs-Rahmen befindet sich ein weiterer Rahmen, in den normalerweise zwei Zeilen ausgegeben werden. Darin stehen Quelltextdatei, Zeilen- und Spaltennummer des gera-

de angewählten Fehlers und noch einmal eine Wiederholung der schon im oberen Rahmen ausgegebenen Fehlermeldung. Unterhalb dieser Anzeige sehen Sie einen Ausschnitt aus dem Quelltext mit einem Pseudo-Cursor an der fraglichen Position.

Wenn allerdings der Linker Fehler meldet, steht anstelle der Sourceposition (was weiß ein Linker schon von Quelltexten!) schlicht **„Linker Error"**, und dort, wo sonst der Quelltext-Ausschnitt dargestellt wird, steht eine eventuell verkürzte Liste der beim Zusammenbinden aufgetretenen Fehler, also hauptsächlich Namen von undefinierten oder gleich mehrfach definierten Linker-Symbolen.

Die eigentliche Fehlermeldungen sind recht knapp gehalten und in der vorliegenden Version obendrein auch noch auf Englisch. Wenn Sie aber eine der Tasten <H> oder <Help> drücken, erscheint ein HotHelp-Fenster mit weitergehenden und deutschsprachigen Informationen. In der Light-Version geht das leider nicht, denn da ist ja kein HotHelp enthalten.

Weniger häufig treten beim Übersetzen unter dem Titel **„Fatal Error"** Fehler bei der Codegenerierung oder beim Assemblieren auf. Meist sind diese auf Speichermangel zurückzuführen, und dementsprechend heißen sie **„Out of Memory"**, **„Workspace overflow"** oder wie auch immer.

Last not least gibt es ganz unten im Compiler-Fenster noch eine Gadgetleiste. Zunächst ist hier nur das Gadget ganz rechts mit der Aufschrift ***„Abbrechen"*** anwählbar. Bei Bedarf werden die übrigen Gadgets aktiviert.

2.4.1.2 Abbruch und Ausstieg

Im Gegensatz zu einfachen Kommandozeilen-Compilern besitzt das MaxonC++-Fenster eine Vielzahl von Knöpfen und Funktionen, an denen der C++-Programmierer herumspielen kann. Alle Gadget-Funktionen sind, um den Bedürfnissen der „echten Programmierer" entgegenzukommen, auch durch Tasten oder Tastenkombinationen bedienbar.

Beim Start des Compilers können Sie im Compiler-Fenster zunächst nur zwei Dinge tun: Das Fenster schließen oder die Übersetzung abbrechen.

Mit einem Klick auf das ***Schließ***-Gadget wird die Übersetzung abgebrochen und das Fenster geschlossen. Die Taste <ESC> hat die gleiche Wirkung.

Außerdem gibt es unten rechts das bereits erwähnte ***„Abbrechen"***-Gadget. Man benutzt es, wenn man den Compiler zwar abbrechen, aber noch irgendwelche im Fenster ausgegebenen Meldungen in Ruhe lesen möchte. Die kanonische Tastenkombination <Control>+<C> entspricht diesem Gadget.

Falls der Compiler aber schon fertig ist, wird das Compiler-Fenster bei einem Klick auf ***„Abbrechen"*** geschlossen.

2.4.1.3 Auswahl von Fehlermeldungen

Laut Murphy geht bei so ziemlich jedem Compiler-Lauf etwas schief, und meist sogar etwas mehr als nur „etwas". Deshalb bietet das Compiler-Fenster Platz für bis zu zehn Fehlermeldungen auf einmal.

Eine dieser jeweils einzeiligen Meldungen ist immer invertiert und somit hervorgehoben, und zu genau dieser Meldung werden unterhalb des Rahmens zusätzliche Informationen angezeigt. Mit einem Mausklick auf eine Fehlerzeile macht man diese zur gerade privilegierten, invertierten Meldung.

Das geht aber auch über die Tastatur: Mit der <Leertaste> durchläuft man die Fehlermeldungen von oben nach unten, und mit <Backspace> in die andere Richtung. Alternativ können Sie auch die Cursortasten <Auf> und <Ab> benutzen.

2.4.1.4 Vom Compiler in den Editor

Der besondere Witz an einer integrierten Umgebung ist ja das perfekte Zusammenspiel von Compiler und Editor, insbesondere das automatische Anspringen von Fehlerstellen.

Das funktioniert bei MaxonC++ denkbar einfach: Sobald eine Fehlermeldung im Compiler-Fenster ausgegeben wird, werden auch zwei Gadgets mit der Aufschrift ***„Anspringen"*** bzw. ***„In Editor"*** aktiviert. Nun wählen Sie sich wie oben beschrieben Ihre Wunsch-Fehlermeldung aus (das geht übrigens auch, während der Compiler noch arbeitet) und klicken ***„Anspringen"*** an. Prompt lädt Edward die zugehörige Quelltextdatei (falls der Text nicht ohnehin schon im Editor vorliegt) und setzt den Cursor an die bemängelte Position. Nun können Sie den Fehler berichtigen - vorausgesetzt natürlich, der Compiler läuft nicht noch und hat ein „Lock" auf diesen Text eingerichtet. Das Compiler-Fenster wird dabei nicht geschlossen, so daß Sie noch weitere Fehlerstellen anspringen können.

Die Integration von Compiler und Editor geht allerdings nicht so weit, daß im Compiler-Window Quelltext-Veränderungen beachtet würden. Wenn Sie z. B. zuerst einen Fehler berichtigen und dabei einige Zeilen einfügen oder löschen müssen, merkt *MCPP* nichts davon. Sollten Sie also anschließend ein weiter unten im Quelltext gelegene Fehlerstelle anspringen lassen, landen Sie gnadenlos in der Zeile mit der Nummer, in der der Fehler lag, als er bemerkt wurde - und durch die inzwischen vorgenommenen Änderungen stimmt diese Zeilennummer inzwischen vielleicht nicht mehr, und Sie landen einige Zeilen zu hoch oder tief. Das dürfte aber in der Praxis kaum ein Problem darstellen, denn im Compiler-Fenster wird immer noch der ursprüngliche Quelltext mit dem Cursor an der richtigen Stelle angezeigt, und durch einen kurzen Textvergleich zwischen Compiler- und Editorfenster können Sie schnell die wirkliche Fehlerposition „zu Fuß" ansteuern. Außerdem ändert man in der Praxis bei Compilerfehlern meist sowieso nur die fragliche Zeile, ohne großartiges Löschen und Einfügen von Zeilen.

Das Gadget ***„Anspringen"*** ist nebenbei bemerkt mit dem Shortcut <A> belegt.

Manchmal tritt während der Übersetzung nur ein einziger Fehler auf, der vielleicht noch den einen oder anderen Folge-Fehler nach sich zieht, aber trotzdem nur eine einzige Änderung am Quelltext notwendig macht. Deshalb gibt es auch noch das Gadget ***„In Editor"***. Hier wird im Editor die gerade angewählte Fehlerposition angesprungen (genau wie beim ***„Anspringen"***-Gadget) und gleichzeitig das Compiler-Fenster geschlossen - wobei natürlich auch der Compiler abgebrochen wird, sollte er noch am arbeiten sein.

Alternativ können Sie auch die Tasten <Return> oder <E> drücken, und ebenso gut können sie auch mit der Maus einen ***Doppelklick*** auf die Fehlermeldung machen.

2.4.1.5 „Weiter"

Treten beim Übersetzen bzw. Linken mehr Fehler auf, als im Rahmen dargestellt werden können (das sind bekanntlich zehn Stück), wartet der Compiler erst einmal ab, und das ***„Weiter"***-Gadget wird anwählbar. Nach einem Klick auf selbiges läuft der Compiler wieder weiter, bis erneut zehn Fehlermeldungen voll sind. Meist ist es aber nicht unbedingt sinnvoll, nach mehr als zehn Fehlern noch weiter zu übersetzen, denn normalerweise kommt dabei nicht mehr so ganz viel sinnvolles 'raus. Deshalb wird man in der Regel das ***„Weiter"***-Gadget meiden und lieber die Fehler korrigieren.

2.4.1.6 Programmstart aus dem Compiler-Fenster

Wenn MaxonC++ bei der Übersetzung Fehler oder Warnungen ausgegeben hat, aber trotzdem ein lauffähiges Programm erzeugen und linken konnte, taucht unten rechts das ***„Starten"***-Gadget auf. Mit einem beherzten Klick darauf können Sie das Programm starten - aber vorher sollten Sie sich genau überlegen, ob die gemeldeten Fehler beim Starten des Programms nicht zu einem Guru oder ähnlich unangenehmen Effekten führen können! Bedenken Sie z. B., daß für eine Programmzeile, die einen groben Fehler enthält, in der Regel keinen Code erzeugt wird - und in einer „Ohne Netz und doppelten Boden"-Programmiersprache wie C++ bzw. C kann auch eine fehlende Anweisung fatale Folgen haben...

2.4.1.7 Erneutes öffnen des Compiler-Fenster

Wenn Sie das Compiler-Fenster geschlossen haben und es doch noch einmal sehen wollen, können Sie es mit dem Menüounkt **"Fehlermeldungen"** noch einmal öffnen. Wie der Name schon andeutet, sind dann alle Fehlermeldungen der letzten Compilierung wieder sichbar, und auch sonst präsentiert sich das Fenster in dem Zustand, in dem es sich beim Schließen befand. Ein Tip für Maus-Hasser: Auch wenn das Compiler-Fenster noch geöffnet ist und der Compiler vielleicht sogar noch arbeitet, können Sie es mit dem Menü - oder besser mit einer sinnvoll gewählten Tastenbelegung - jederzeit aktivieren und in den Vordergrund holen. So können Sie zwischen Editor- und Compiler-Fenster hin- und herwechseln, ohne zur Maus greifen zu müssen.

2.4.2 Ausführen und Abbrechen

Die zweite wichtige Funktion aus dem **„Compiler"-Menü** heißt **„Ausführen"** und tut auch genau das: Ein C++-Programm wird gestartet.

Vorher prüft *MCPP* allerdings noch, ob nicht vielleicht erst noch der Compiler konsultiert werden muß, und startet selbigen automatisch.

Anschließend wird ein Text-Fenster für die Ein- und Ausgaben des Programms geöffnet. Falls das Programm tatsächlich etwas ausgegeben hat, erscheint nach erfolgreicher Programmausführung die Aufforderung **„Press Return"** - Sie sollen also die <Return>-Taste drücken und so bestätigen, daß Sie die Ausgaben des Programms auch gelesen haben.

Während das Programm läuft, sind verschiedene Funktionen des Systems gesperrt, z. B. darf man es (trotz Multitasking) nicht ein zweites mal gleichzeitig starten, oder man kann den Compiler nicht laufen lassen, weil man dem laufenden Programm ja dadurch den Code unter den Füßen wegziehen bzw. überschreiben würde. Das ist sicher sinnvoll und stört wohl nicht weiter - es sei denn, das Programm ist irgendwo hängengeblieben und läßt sich nicht normal beenden.

Gegen „normale" Endlosschleifen hilft die Compileroption ***„Unterbrechen"***. Schwieriger wird es dagegen, wenn der Prozeß abgestürzt ist, auf ein Ereignis wartet, das nie eintreten kann oder die erwähnte Option ***„Unterbrechen"*** einfach nicht benutzt wurde. Deshalb (und nur für solche Fälle!) gibt es im **Compiler-Menü** die Funktion **„Abbrechen"**, die den gestarteten Prozeß schlicht abtötet und dann das Ausgabe-Fenster schließt. Auf den ersten Blick eine praktische Lösung, aber leider gibt es auf dem Amiga (im Gegensatz z. B. zu Unix) keine saubere Möglichkeit, einen Prozeß aus dem System zu entfernen. Insbesondere werden dabei keine Ressourcen (reservierter Speicher, geöffnete Dateien, Screens und Fenster etc.) freigegeben. Sie müssen davon ausgehen, daß sie auf diese Weise freien Speicher verlieren, wenn Sie einen Prozeß auf diese brutale Art abbrechen. Aber das ist im Zweifel immer noch angenehmer, als das MCPP-System neu zu starten oder gar den Rechner zu booten.

2.4.3 Aufruf des Debuggers

Im Compiler-Menü finden Sie auch die Funktion **„Debugger"**. Wenn Sie Besitzer der Developer-Version sind, können Sie hier den Debugger *MaxonSDB* aufrufen.Um den Speicherverbrauch des MaxonC^{++} Systems nicht unnötig zu vergrößern, wurde der Debugger nicht wirklich in die Entwicklungsumgebung integriert, sondern muß bei Bedarf nachgeladen werden. Im „System"-Einstellfenster können Sie die dafür notwendigen Einstellungen vornehmen, falls Sie mit den Defaultwerten nicht zufrieden sind.

Bei Aufruf des Menüpunkts **„Debugger"** prüft *MCPP* zunächst, ob bereits eine ausführbare Programmdatei geschrieben wurde - falls nicht, geschieht dies jetzt automatisch. Dabei wird auch auf jeden Fall eine *„.mdbg"-Datei* erzeugt, aus der der Debugger erfährt, aus welchen Objektdateien das Programm besteht. Die ebenfalls wichtigen *„.mdbi"-Dateien* können jetzt aber nicht mehr erzeugt werden, da sie nur während der Übersetzung geschrieben werden können. *MaxonSDB* wird Ihnen das Fehlen solcher Dateien ggf. melden. Mein Tip: Wenn Sie den Debugger besitzen und prinzipiell auch benutzen möchten, sollten sie die Option für die Erzeugung der Debugger-Dateien generell eingeschaltet lassen.

2.4.4 Exe- und Objektdateien

Ungeachtet allen damit verbundenen Komforts werden Sie ihre Programme wohl nicht immer nur aus der integrierten Umgebung starten wollen. Deshalb gibt es im **„Compiler"-Menü** zwei Funktionen, die das übersetzte Programm in Dateien schreiben:

☞ **„Exe-Datei"** erzeugt eine ausführbare Programmdatei - vorausgesetzt natürlich, ein Programm ist korrekt übersetzt und gelinkt worden.

☞ **„Objekt-Datei"** schreibt zur zuletzt compilierten Übersetzungseinheit eine Objektdatei, aus der man später z. B. mit dem `"mcppc"` ein ausführbares Programm erzeugen kann. Im „Make"-Modus ist diese Funktion nicht unbedingt sinnvoll, denn dort werden diese Objektdateien ja beim Compilieren automatisch erzeugt.

2.4.5 Daten und Informationen

Mit dem Menüpunkt **„Information"** öffnet man ein Fenster, in dem Informationen über den Status von *MCPP* angezeigt und ständig aktualisiert werden. Neben dem üblichen Rhabarber über Autorschaft und Copyright (klicken Sie doch einfach mal auf den Namen des Autors...) können Sie hier bei Bedarf nachlesen, welchen Quelltext Sie zuletzt übersetzt oder welche Make-Datei gerade geladen ist. Liegt ein übersetztes Programm im Speicher vor, wird auch seine Größe angegeben. Dabei handelt es sich genaugenommen um den Platzbedarf des Programms, wenn es in den Speicher geladen wird, und nicht um die Größe der Exe-Datei.

2.5 Features und Optionen

2.5.1 Ein kurzer Überblick

Für viele Programmierer ist das Schönste an einem Compiler, daß man so viele Optionen und Switches hat, an denen man herumspielen kann. Bei *MCPP* nimmt man sämtliche dieser Einstellungen über Dialoge vor, deren es genau drei gibt: Einen – **„System"** – für allgemeine Einstellungen, die in irgendeiner Weise mit dem MCPP-System zu tun haben, einen weiteren für die unvermeidlichen **Compiler-Optionen** und schließlich einen eigenen Dialog für die **Linker-Steuerung**.

Wie Sie sich vielleicht denken können, beherrschen *MCPP* und die Kommandozeilen-Version *„mcppc"* praktisch die selben Einstellungen und Optionen, nur die Bedienung ist eben unterschiedlich. Deshalb sollten Sie sich dieses Kapitel auch dann durchlesen, wenn Sie ausschließlich den *„mcppc"* benutzen wollen, denn bei der Beschreibung der Optionen im *„mcppc"*-Kapitel wird hauptsächlich das „Wie" der Einstellungen beschrieben, während ausschließlich hier ausführlich erläutert wird, was diese Optionen genau bedeuten.

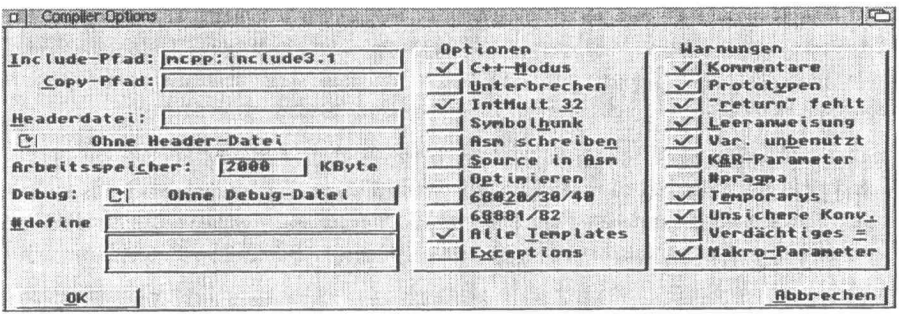
Zurück zum *MCPP*: Wenn Sie wollen, können Sie alle drei Dialoge gleichzeitig öffnen. In jedem der Fenster finden Sie am unteren Rand die obligatorischen Gadgets ***„OK"*** und ***„Abbruch"***, mit denen Sie die vorgenommenen Einstellungen bestätigen bzw. Sie wieder verwerfen können.

2.5.2 Compiler-Optionen

2.5.2.1 Include-Dateien

Wie eigentlich alle modernen Programmiersprachen, sind C und C++ ziemlich „dumm": es gibt für den Compiler keinerlei vordefinierte Funktionen (während z. B. in Pascal `"WriteLn"` oder in BASIC `"print"` stets zur Verfügung stehen). Vielmehr stehen Definitionen für die Standard-Funktionen in diversen Includedateien, die man explizit einbinden muß.

Vielleicht wissen Sie schon, daß es dafür zwei verschiedene Möglichkeiten gibt: Schließt man den Dateinamen `"so"` in Anführungszeichen ein, sucht der Compiler (oder der Preprozessor, um genauer zu sein) vom aktuellen Verzeichnis aus, während die `<so>` in spitze Klammern gesetzten Dateinamen Standard-Includedateien bezeichnen und in einem ganz bestimmten Verzeichnis gesucht werden. Von genau diesem Verzeichnis, und wie man es festlegt, soll hier die Rede sein.



Im **"Compiler"**-Dialogfenster legen Sie mit den beiden Textgadgets oben links genau dieses Verzeichnis fest. Wenn Sie nur das obere Gadget benutzen, dürfte die Funktionsweise klar sein: Der Compiler sucht die Include-Dateien im dort angegebenen Verzeichnis. Für alle, die sich um den RAM-Bedarf keine allzu großen Sorgen machen müssen, gibt es auch noch die Möglichkeit, Includes umkopieren und dabei leicht vorcompilieren zu lassen. Dazu dient das zweite Stringgadget: Hier können Sie ein zweites Verzeichnis angeben, das vorzugsweise in der *"RAM:"*-Disk liegen sollte, aber natürlich nicht muß. Der Compiler sucht dann zuerst in diesem zweiten Verzeichnis nach den Includedateien. Findet er sie dort nicht, greift er auf das andere Verzeichnis zu und kopiert die benutzten Dateien von dort automatisch um. Dabei werden sie aber auch noch leicht „vorcompiliert" (nämlich als Token abgespeichert), wodurch alle weiteren Übersetzungen etwas beschleunigt werden. Durch Angabe eines zweiten Pfades können Sie die Dateien also gleichzeitig auf ein schnelleres Speichermedium umkopieren und vorcompilieren lassen.

Übrigens legt MaxonC^{++} vor dem Umkopieren/Vorcompilieren bei Bedarf selbständig die benötigten Unterverzeichnisse an.

Vor einer kleinen Heimtücke des Systems sei noch gewarnt: Achten Sie bitte darauf, daß Sie weder an den Anfang noch an das Ende der Dateinamen versehentlich irgendwelche Leerzeichen anhängen! *MCPP* versucht sonst gänzlich ungerührt, auf die angegebenen Pfade einschließlich der überflüssigen Spaces zuzugreifen, und da wird er naturgemäß nicht allzu viel finden. Außerdem sollten Sie sich, wenn Sie in die RAM-Disk vorcompilieren lassen, nach dem Verlassen des MCPP-Systems auch nicht über einen scheinbaren Speicherverlust wundern - das liegt einzig und allein an den vorcompilierten Daten in der RAM-Disk, die sich bei Bedarf leicht mit einem CLI-Befehl wie

```
delete ram:include all quiet
```

löschen lassen.

2.5.2.2 Vorcompilierte Header-Dateien

MaxonC++ kann Teile von Programmen vorcompilieren, d. h. an einer vom Benutzer frei wählbaren Stelle kann der Compiler alle seine internen Daten - und damit die bisher gelesenen Deklarationen und Definitionen - in einer Datei abspeichern. Beim nächsten Compilerdurchlauf kann diese Datei sehr schnell geladen und so viel Zeit gespart werden.

Dieses Feature ist dafür gedacht, die Programmentwicklung zu beschleunigen, wenn sehr viele Header-Dateien benutzt werden, z. B. die AmigaOS-Includes oder die MaxonC++ Klassenbibliothek "*EasyObjects*". Damit der Compiler aber weiß, wo die Header aufhören und Ihr eigenes Programm anfängt, muß man ihm diese Stelle mit einer Zeile `"#pragma header"` markieren.

Am besten schreiben Sie in den Quelltext jeder Überstzungseinheit zunächst die `"#include's"` all jender Headerdateien, die Sie nicht selbst geschrieben haben oder an denen sich aus anderen Gründen nichts mehr ändern wird. Dann folgt die Zeile `"#pragma header"`, und anschließend folgen die Include-Anweisungen für Ihre eigenen Header-Dateien und der Rest Ihres Programms.

Normalerweise hat `"#pragma header"` keinerlei Wirkung. Wenn Sie aber im Textfeld "Header-Datei" des Compiler-Dialogfensters einen beliebigen Dateinamen eintragen und das Gadget darunter auf "Header-Datei schreiben" stellen, hält der Compiler, sobald das Pragma erreicht wird, kurz inne und schreibt alle Definitonen und Deklarationen in die namgegebene Datei. Danach arbeitet er weiter, als sei nichts geschehen.

Vor dem nächsten Compilerdurchlauf schalten Sie dann einfach um auf "Header-Datei lesen", wobei Sie den Dateinamen unverändert belassen. Dann liest der Compiler die vorcompilierten Header aus der angegebenen Datei, sucht im Quelltext nach dem `"#pragma header"` und setzt dort mit seiner Übersetzungstätigkeit ein.

Da alles, was vor dem `"#pragma header"` steht, komplett ignoriert wird, muß das Pragma folglich in der Haupt-Quelltextdatei der Übersetzungseinheit - und nicht etwa in einer Includedatei - stehen.

Denken Sie bitte daran, daß *MCPPC* nicht prüft, ob der Inhalt der vorcompilierten Header-Dateien überhaupt noch mit dem ursprünglichen Quelltext übereinstimmt. Deshalb empfehle ich noch einmal dringend, nur solche Header-Dateien vorzucompilieren, an denen sich während der Programmentwicklung nichts mehr ändert.

Wundern Sie sich bitte auch nicht, wenn er Compiler vor dem eigentlichen Schreiben der Header-Dateien eine ganze Weile scheinbar überhaupt nichts tut. In dieser Zeit sammelt und sortiert er seine Daten, und das kann dauern. An Speicher zieht er sich dabei auch so einiges rein, so daß Sie bei knappem RAM-Ausbau und/oder großen Header-Files eventuell die integrierte Umgebung verlassen und auf den Shell-Compiler "*mcppc*" ausweichen müssen. Beim späteren Laden der so erzeugten Header-Dateien werden Sie aber keine derartigen Probleme haben, denn gegenüber der "normalen" Übersetzung spart man mit vorcompilierten Headern nicht nur eine memge Zeit, sondern auch noch ein bißchen Speicherplatz.

2.5.2.3 Der Arbeitsspeicher des Compilers

Hin und wieder braucht der MaxonC++ Compiler einen zusammenhängenden Speicherbereich, nämlich bei der Codegenerierung und Assemblierung. Außerdem legt er nach erfolgreichem Linken in diesem Speicherblock eine lauffähige Kopie des erzeugten Programms ab.

Was Sie damit zu schaffen haben?

Eigentlich nichts, aber leider weiß MaxonC++ nicht im Voraus, wieviel Speicher er dafür brauchen wird. Der Compiler kann später höchstens einen Überlauf feststellen, aber vergrößern kann er diesen ***„Workspace"*** genannten Speicherblock dann nicht mehr, weshalb er mit einer Meldung wie **„Workspace Overflow"** aussteigt.

Deshalb müssen Sie leider persönlich festlegen, wie groß dieser Workspace sein soll, und zwar mit den entsprechendem Integer-Gadget im Compiler-Requester. Der Defaultwert von 80 kByte dürfte übrigens für die allermeisten Zwecke ausreichen, und wenn Sie mit RAM etwas knapp sind, können Sie ruhig versuchen, die Workspace-Größe weiter zu verringern. Sollte der Compiler dagegen einmal **„Workspace Overflow"** melden, setzen Sie die Größe einfach etwas herauf und versuchen es noch einmal.

2.5.2.4 Debug-Datei

Das Gadget ***"Debug"*** ist für Sie nur von Interesse, wenn Sie den MaxonSDB besitzen. Der Compiler kann zu jeder Übersetzungseinheit eine *"MDBI"*-Datei anlegen, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

Diese können aber "mehr unfangreich" oder "weniger umfangreich" sein: Wenn Sie sich für die Einstellung ***"Komplette Debug-Datei"*** entscheiden, taucht praktisch jede Deklaration des Programms in der Datei auf und steht dann auch im Debugger zur Verfügung. Bei einem Programm, das z. B. die unvermeidlichen AmigaOS-Includes enthält, tauchen dann die ganzen Betriebssystem-Strukturen in der *MDBI*-Datei auf - und das bläht diese Dateien mächtig auf.

Da man sich beim Debugging normalerweise mehr für sein eigenes Programm als für Interna des Betriebssystems oder irgendwelcher Klassebibliotheken interessiert, sollte man sich besser für die "kleine" Variante der *MDBI*-Dateien entscheiden. Dann werden Deklarationen aus Standard-Includedateien (das sind die, die man mit den `<spitzen Klammern>` includiert) nicht an den Debugger weitergereicht. Das spart Zeit und Speicherplatz, und zwar gleichermaßen beim Compilieren des Programms wie auch bei der Debugger-Sitzung.

2.5.2.5 Vordefinieren von Symbolen

Mit den drei Gadets, die mit ***„#define"*** überschrieben sind, können Sie bis zu drei Preprozessor-Symbole vordefinieren. Jeder Eintrag in einem dieser Textgadgets wirkt dann so, als stände er mit einem vorangestellten `"#define"` am Anfang jedes übersetzten Quelltexts. Dieses Feature entspricht also der `"-d"`-Option von „mcppc".

Ein Beispiel: Sie haben als vorsichtiger Programmierer die Includedatei `"<assert.h>"` eingebunden und diverse Sicherheitsabfragen in Ihr Programm gesetzt - übrigens ein löbliches Unterfangen.

Um diese Abfragen automatisch aus dem Programm auszublenden, müssen Sie dann `"#define NDEBUG"` irgendwo an den Anfang des Quelltexts setzen - oder ganz einfach `"NDEBUG"` in eines der drei *„#define"*-Gadgets schreiben.

Sie können Symbolen aber nicht nur „einfach so" definieren, sondern ihnen auch Werte zuweisen. Ein Gadget-Eintrag wie `"N 100"` definiert das Symbol `"N"` als `"100"`. Leider akzeptiert der Compiler bei solchen Vorab-Definitionen immer nur ein Token pro Symbol: Der Eintrag `"ANZAHL (17+4)"` belegt `"ANZAHL"` mit dem Zeichen `"("` und ignoriert den Rest einfach. Aber komplexe Definitionen wird man normalerweise ohnehin in den Quelltext schreiben, während derartige „externe" Symboldefinitionen eher für irgendwelche Flags wie `"NDEBUG"` benutzt werden.

2.5.2.6 Diverse kleine, aber trotzdem wichtige Schalter

Nahezu unübersehbar befinden sich im Compiler-Dialog zwei Spalten mit diversen Switches - also so etwas wie ein virtuelles Mäuseklavier. Die linke Spalte ist mit *„Optionen"* überschrieben und enthält demzufolge eine mehr oder weniger geordnete Ansammlung von ein- und ausschaltbaren Compileroptionen.

2.5.2.6.1 C++

Mit dem Gadget *„C++"* können Sie wählen, ob MaxonC++ als echter C++-Compiler oder kastriert im ANSI C-Modus arbeiten soll. Ich hoffe doch stark, daß Sie die C++-Option nur in Ausnahmefällen ausschalten werden. Sie können diese Umschaltung übrigens auch mit einer `"#pragma"`-Zeile direkt im Quelltext vornehmen. Näheres dazu erfahren Sie im Kapitel über den Preprozessor (5.5.1 im Tutorial).

2.5.2.6.2 Unterbrechen

Mit *„Unterbrechen"* legen Sie fest, ob der Compiler im erzeugten Code Breakpoints setzen soll. Wenn diese Option angewählt ist, baut er in jede Schleife eine Abfrage der Tastenkombination <Ctrl>+<C> ein.

2.5.2.6.3 IntMult 32

Der 68000er-Prozessor kann leider keine Langworte multiplizieren und dividieren, sondern nur zwei 16-Bit-Werte zu einem 32-Bit Ergebnis multiplizieren und eine 32-Bit-Zahl durch ein 16-Bit-Wort dividieren, wobei Quotient und Rest jeweils 16 Bit breit sind. Für numerische Anwendungen ist das natürlich nicht so schön, weshalb MaxonC++ optional solche Operationen durch Bibliotheksfunktionen ersetzen kann.

Diese korrekt rechnenden Funktionen werden benutzt, wenn das Gadget *„IntMult 32"* aktiviert ist, und kosten natürlich einiges an Rechenzeit. Dieser Laufzeit-Aufwand ist aber in der Praxis nicht so gravierend, und für korrekte Ergebnisse ist uns schließlich nichts zu teuer.

Falls Sie aber die 68020-Option benutzen, ist der Schalter *„IntMult 32"* bedeutungslos.

2.5.2.6.4 Debug-Datei

Die Compileroption ***„Debug-Datei"*** ist für Sie nur von Interesse, wenn Sie den Maxon Debugger besitzen. Der Compiler legt dann zu jeder Übersetzungseinheit eine Datei an, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

2.5.2.6.5 Symbol-Hunk

Anstelle eines Source-Level-Debuggers wie dem hochkomfortablen MaxonSDB kann man natürlich auch einen symbolischen Debugger wie *„MetaScope"* benutzen. Um dabei überhaupt etwas wiederzufinden, sollte man sich bei Exe- und Objektdateien sogenannte ***„Symbol-Hunks"*** erzeugen lassen. Genau dafür dient die gleichnamige Option.

Wichtig ist, daß Sie ***„Symbol-Hunk"*** schon beim Compilieren und nicht erst beim Erzeugen der Exe-Dateien anwählen, um den Compiler zu veranlassen, die notwendigen Informationen zu sammeln und aufzubewahren.

2.5.2.6.6 Asm schreiben

Wenn ***„Asm schreiben"*** angewählt ist, erzeugt der Compiler zu jedem Text, den er erfolgreich übersetzt, ein Assembler-Listing. Der Assembler-Quelltext wird in eine Datei geschrieben, deren Name wie der des C-Quelltexts anfängt und mit *„.s"* endet, also z. B. *„ballo.s"* zu *„ballo.c"*. Wenn Sie schon andere Compiler näher kennen, wundern Sie sich vielleicht darüber, daß der Compiler scheinbar einen so optimierten Assembler-Text erzeugt, daß für den Assembler nichts mehr zum Optimieren übrig bleibt. Auch werden Sie sich vielleicht fragen, warum Sie mit dem integrierten Assembler von MaxonC++ nicht selbstgeschriebene Assembler-Programme übersetzen lassen können.

In beiden Fällen handelt es sich um Fehlschlüsse, denn MaxonC++ ist etwas ungewöhnlich implementiert. Der Compiler erzeugt überhaupt keinen Assembler-Quelltext, und es gibt auch keinen Assembler.

Um diese kryptische Bemerkung etwas zu erhellen: Der Compiler erzeugt bei der Codegenerierung einen Zwischencode in einem internen binären Format, das zwar in gewisser Weise wie ein Assembler-Quelltext aufgebaut ist, sonst aber absolut nichts mit einer ASCII-Textdatei zu tun hat. Der „Assembler" ist dann ein ziemlich dummes kleines Prográmmchen, daß diesen Zwischencode in echten Maschinencode übersetzt und dabei noch ein paar kleine Optimierungen durchführt. Diese Low-Level-Optimierungen, zu denen z. B. Short Branches gehören, trägt der Assembler dann auch immer gleich im Zwischencode ein, damit er beim nächsten Pass dasselbe nicht noch einmal optimieren muß.

Es sei noch einmal darauf hingewiesen, daß dieser „Assembler" wirklich ein ziemlich dummes Programm ist, das z. B. nur eine Teilmenge der 680x0-Assemblersprache beherrscht und sich stets darauf verläßt, daß der „Quelltext" - also dieses interne Format, das ihm als Eingabe dient, fehlerfrei ist. Somit ist dieser Wannabe-Assembler absolut NICHT IN DER LAGE, NORMALE ASSEMBLER-QUELLTEXTE ZU VERARBEITEN. Und dann gibt es noch einen Programmteil, der noch viel simpler als der „Assembler" ist und den internen Zwischencode in lesbaren Assembler-Quelltext umwandelt. Diese

Funktion wird benutzt, wenn die Option ***„Asm schreiben"*** angewählt ist, und da die Datei geschrieben wird, nachdem der Pseudo-Assembler seine Arbeit getan hat, sind darin also auch schon alle Optimierungen vorgenommen. Der von MaxonC++ generierte Assembler-Source entspricht also genau dem erzeugten Code - Sie können sich den Reassembler sparen, wenn sie die Codequalität genauer unter die Lupe nehmen wollen.

2.5.2.6.7 Source in Asm

Den oben beschriebenen Quelltext kann man sich etwas übersichtlicher gestalten lassen, indem man zusätzlich die Option ***„Source in Asm"*** wählt. Dann wird nämlich der C-Quelltext an jeweils passender Stelle in den Assembler-Source als Kommentar eingefügt, so daß sich die Bedeutung der endlosen Assembler-Listings dem menschlichen Leser einfacher erschließt.

2.5.2.6.8 Optimieren

Guter Code ist mit einem gewissen Aufwand verbunden, sowohl in Bezug auf den Speicherplatzbedarf des Compilers als auch auf die Übersetzungszeit. Deshalb gibt es die wahlweise ab- oder zuschaltbare Option ***„Optimieren"***, die MaxonC++ veranlaßt, die Codeerzeugung etwas schlauer (und teurer) aufzuziehen.

Das Thema ***„Optimieren"*** ist nahezu unerschöpflich, und folglich dürfte bei jeder neuen Version von MaxonC++ das eine oder andere Feature hinzukommen. Im Moment macht der Compiler bei aktiver Optimier-Option mindestens folgendes:

- ☞ Alle anderweitig entbehrlichen Register werden für lokale Variablen benutzt. Die Entscheidung, welche Variablen in Register geladen werden sollen, fällt über gewichtetes Zählen ihres Auftretens.
- ☞ Wenn sich herausstellt, daß eine Funktion keine lokalen Variablen auf dem Stack mehr benötigt (z. B. weil alle in Register umgelegt werden konnten), wird das Einrichten eines Stack-Rahmens mittels `"LINK"` und `"UNLK"` unterdrückt.
- ☞ Der Assembler führt nicht nur einen, sondern zwei optimierende Passes durch. Damit werden z. B. auch Sprünge nach vorn zu Short-Branches optimiert.

Diverse Mini-Optimierungen, die keinen besonderen Aufwand erfordern, werden natürlich immer durchgeführt und sind nicht abschaltbar. Die Grenze zwischen „vermiedenem schlechtem Code" und einer „Optimierung" ist bekanntlich fließend.

2.5.2.6.9 Code für höhere Prozessorgenerationen

Die bewährten MC68000-Prozessoren sind in der Amiga-Welt eine aussterbende Spezies. 32Bit-Prozessoren (68020 bis 68060) und Floating Point Units (68881 und 68882) setzen sich mehr und mehr durch. Damit stehen dem Programmierer neue, leistungsfähige Maschinenbefehle zur Verfügung, die teilweise eine erstaunliche Geschwindigkeitssteigerung bringen (das ganze ist allerdings nicht so dramatisch wie in der INTEL-Welt, wo die „schnellen" 468er-DOS-PC's wegen der Zwangskompatiblität zum 8-Bit-Prozessor 8088 meist nur einen Teil ihrer Möglichkeiten nutzen können - aber lassen wir das).

Die 32-Bit-Prozessoren bieten jedenfalls insbesondere die lange vermißten Operationen für Langwort-Multiplikation und -Division und einige nette Features für die Handhabung von Arrays und Bitfeldern. Mit dem Switch ***„68020/30/40"*** veranlassen Sie den Compiler, dies auch zu nutzen. Aber Achtung: Ihr Code ist dann natürlich auf einem altem Amiga 500 wahrscheinlich nicht mehr lauffähig und schmiert an unerklärlicher Stelle plötzlich ab. Und allzu drastische Geschwindigkeitssteigerungen sollten Sie von dieser Option nicht erwarten.

Bei Programmen, die viel mit Fließkommazahlen rechnen, bringt die Anschaffung einer FPU allerdings durchaus beachtliche Performancezuwächse. Die Option ***„68881/82"*** nutzt bei der Codeerzeugung diese Möglichkeit aus.

2.5.2.6.10 Strategien für Template-Funktionen

Templates sind ein ausgesprochen nützliches und leicht verständliches neues Sprachkonstrukt, leider aber nicht ganz einfach zu implementieren. Vor allem die Funktionstemplates haben einige überraschende Eigenschaften.

Man kann im allgemeinen nicht sagen, ob die Definition eines Templates richtig oder falsch ist. Stellen Sie sich beispielsweise einmal eines der beliebten Funktionstemplates für einen Sortieralgorithmus (z. B. Quicksort) vor. Der Elementtyp des zu sortierenden Vektors ist dabei üblicherweise ein Templateargument. Wenn nun die Funktion Vergleichsoperatoren wie `"<"` oder `">="` verwendet, kann aus dem Template nur für solche Typen, auf denen diese Operatoren definiert sind, eine Funktion generiert werden. Der Programmierer kann dem abhelfen, indem er die benötigten Funktionen für andere Datentypen „von Hand" implementiert. Da man bei einem größeren Projekt aber meist mehr als eine Übersetzungseinheit hat, kann der Compiler nicht wissen, welche der nicht erzeugbaren und trotzdem benötigten Funktionen irgendwo im Gesamtprojekt bereits fertig vorliegen.

Aus dieser Misere behilft sich MaxonC^{++} mit zwei verschiedenen Strategien, aus denen Sie mit der Option ***„Alle Templates"*** aus dem Compiler-Dialogfenster auswählen können:

Wenn die Option angewählt ist, unterstellt der Compiler, daß der Programmierer sich nicht die Mühe gemacht hat, irgendwelche Templatefunktionen von Hand zu programmieren. Deshalb erzeugt er alle Templatefunktionen, die in der vorliegenden Übersetzungseinheit benötigt werden und dort sonst nicht definiert sind. Tritt dabei ein Fehler auf, so wird er gemeldet. Funktionen, die dabei überflüssigerweise erzeugt wurden, werden später vom Linker 'rausgeschmissen.

Eine andere Strategie wird verfolgt, wenn die Option ***„Alle Templates"*** nicht angewählt ist. Auch dann werden „auf Verdacht" alle in Frage kommenden Templatefunktionen generiert. Wenn dabei in einer Funktion aber ein Fehler gefunden wird, wird der Versuch verworfen. Es wird keinerlei Code für die fehlerhafte Funktion erzeugt, dafür wird der Programmierer aber auch nicht mit der Fehlermeldung belästigt. Der Compiler nimmt hier also an, daß alle nicht fehlerfrei erzeugbaren Templatefunktionen irgendwo im Gesamtprojekt sinnreich definiert wurden.

Während der Programmentwicklung rate ich Ihnen dringend, die fragliche Option zunächst einmal eingeschaltet zu lassen. Dies ist nämlich Voraussetzung dafür, daß Sie überhaupt eine aussagekräfti-

ge Meldung über Fehler in einer Templatefunktion erhalten (andernfalls teilt Ihnen lediglich der Linker lapidar mit, daß die Funktion offenbar nicht erzeugt werden konnte, und zwar ohne Angabe von Gründen oder Fehlerstellen). Wenn Sie aber mit fertigen Template-Bibliotheken arbeiten oder ein bereits vorhandenes Programm portieren möchten, kann es irgendwann notwendig werden, die Zwangsgenerierung aller Templatefunktionen abzustellen.

Falls Sie sich jetzt fragen sollten, wieso Sie sich mit derartigen Strategie-Auswahlen herumplagen müssen, so verrate ich Ihnen jetzt, wie eine korrekte Implementierung von Funktionstemplates genaugenommen vorgehen müßte:

- Bei der ersten Übersetzung werden überhaupt keine Templates erzeugt.
- Der Linker versucht, das Projekt zusammenzubinden, und erstellt dabei eine Liste aller undefinierten Funktionen.
- Im finalen Compilerdurchlauf werden alle Module, die irgendwie in dem Ruch stehen, eine hilfreiche Templatedefinition zu enthalten, erneut compiliert. Dabei kann anhand der vom Linker erstellten Liste sicher entschieden werden, welche Funktionen generiert werden müssen.

Eine solche Implementierung wäre nicht nur sehr aufwendig, sondern auch enorm langsam, da jedesmal diverse Module doppelt compiliert werden müßten. Deshalb greift MaxonC++ auf die Methode mit der Strategiewahl durch den Programmierer zurück.

2.5.2.6.11 Exceptions

Wie in Kapitel 8.5 des Tutorials dargestellt, hat es manchmal Vorteile, wenn man ein "klassisches" C++-Programm, das kein Exception Handling benutzt, in einem speziellen Compilermodus übersetzen läßt. Dann werden die mit der Ausnahmebehandlung zusammenhängenden Schlüsselwörter (`"catch"`, `"throw"` und `"try"`) nicht mehr erkannt, und im erzeugten Code wird nicht über notwendige Destruktoraufrufe Buch geführt. Wenn der Schalter "***Exceptions***" ausgeschaltet ist, ist also keine Ausnahmebehandlung möglich. Der Zugewinn an Geschwindigkeit und Kürze ist aber meist gering.

2.5.2.7 Optionale Warnungen und Meldungen

Diverse Sprachkonstrukte sind zwar an sich syntaktisch wie semantisch korrekt, aber trotzdem etwas obskur und möglicherweise Folge eines Versehens des Programmierers. Deshalb gibt es elf Warnungen, die man je nach Situation und persönlichem Programmierstil einzeln ein- und ausschalten kann. In MaxonC++ dienen dazu die Switches in der rechten Spalte des Compiler-Dialogs.

Im Einzelnen gibt es die folgenden Warnungen:

- ***„Kommentare"***: Bei verschachtelten Kommentaren, wie `"/* Test /* Nahu? */"` wird gewarnt. Hier liegt der Verdacht nahe, daß am Ende eines Kommentars einfach das `"/*"` vergessen wurde.
- ***„Prototypen"***: Im C-Modus wird gewarnt, wenn eine Funktion aufgerufen wird, zu der kein Prototyp deklariert wurde. In C++ ist das bekanntlich sowieso ein Fehler.

- *„'return' fehlt"*: Wenn in einer Funktion, deren Ergebnistyp nicht `"void"` ist, nirgendwo ein `"return"` auftaucht, ist sicher etwas faul. Deshalb sollte diese Warning immer eingeschaltet sein.

- *„Leeranweisung"*: Eine Anweisung wie `"42;"` hat keinerlei Wirkung und war folglich wahrscheinlich nicht so gemeint. Ein besonders beliebter Fehler ist auch der Aufruf einer parameterlosen Funktion ohne Argumentliste, etwa `"test;"` statt `"test();"`.

- *„Var. unbenutzt"*: Hier wird gewarnt, wenn eine Variable im automatischen Speicher deklariert, aber nie benutzt wird (dann ist sie wohl überflüssig), oder wenn sie benutzt, aber nirgendwo initialisiert wird (das ist wohl ein Fehler).

- *„K&R-Parameter"*: Das „K&R" steht für „Kernighan & Ritchie" und damit für den Vor-ANSI-Standard, der übrigens keiner war. Mit dieser Option kann man sich bei Parameterlisten im alten Stil eine Warnung ausgeben lassen.

- *„#pragma"*: Es ist natürlich Sinn und Zweck eines `"#pragma"`, daß der Compiler darin eben keine Fehler meldet und alles ignoriert, was er nicht kennt. Wenn Sie aber nicht bloß fremde Quelltexte übernehmen, sondern mit MaxonC^{++} selbst entwickeln, ist es sicher sinnvoll, wenn Sie sich auf unbekannte Pragma's hinweisen lassen - schließlich könnte es sich dabei ja auch ganz einfach um einen Tippfehler handeln.

- *„Temporarys"*: Laut C^{++}-2.0-Standard ist es ein Fehler, wenn eine Referenz auf einen nichtkonstanten Typ mit einem Nicht-L-Wert initialisiert wird und dabei ein temporäres Objekt eingeführt werden muß. Da diese Regel im 1.0-Standard noch nicht existierte, gibt es dafür keine „echte" Fehlermeldung, sondern eine abschaltbare Warnung.

Ein Beispiel:

```
void dup(int &ir)
{ ir *= 2; }

void main()
{ int i;    dup(i);        // OK
long l;  dup(l);           // VORSICHT! }
```

Da hier `"l"` von `"int"` nach `"long"` konvertiert werden muß (in MaxonC^{++} übrigens eine leere Operation, denn intern sind beides 32Bit-Zahlen), wird ein temporäres Objekt eingeführt, und der Programmierer erlebt höchstwahrscheinlich eine Überraschung, denn beim zweiten `"dup"`-Aufruf gibt es keinen Seiteneffekt auf das Argument. Die neue C^{++}-Regel ist also ausgesprochen sinnvoll.

- *„Unsichere Konv."*: Diese Option warnt bei impliziten Typumwandlungen (also ohne Cast) von ganzzahligen in Fließkommatypen.

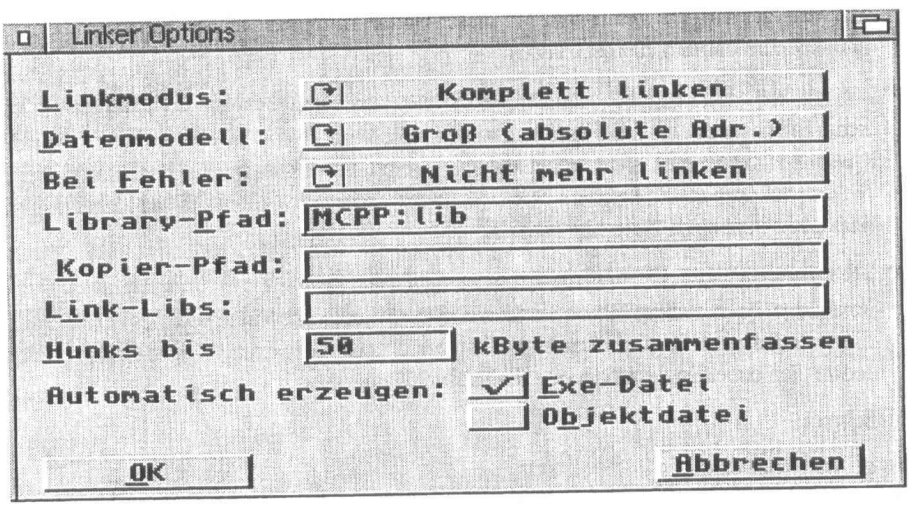
- *„Verdächtiges =="*: Dabei handelt es sich um das beliebte Konstrukt `"if (a=b)..."` - korrekt und bei vielen Programmierern beliebt, aber auch ein typischer Anfängerfehler statt `"if (a==b)"`. Wenn die entsprechende Option aktiv ist, wird jeweils Warnung ausgegeben, wenn

ein `"="` auf diese Weise in einem logischen Ausdruck auftritt, also als Bedingung in `"if"`, `"while"`, `"for"` oder `"do"` sowie als Operand von `"||"`, `"&&"` oder `"!"`.

- ***„Makro-Parameter"***: Wenn ein Argument für ein Preprozessor-Makro sich über mehr als 8 Zeilen oder 200 Token erstreckt, liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde. Damit der Programmierer (das sind Sie) dann nicht endlos suchen muß, gibt es diese optionale Warnung.

2.5.3 Der Linker-Requester

Mit dem Menüpunkt **„Linker"** aus dem **„Optionen"-Menü** öffnen Sie den Linker-Dialog. Hier finden Sie diverse Optionen zur Steuerung des Linkers.



2.5.3.1 Datenmodelle

Die Umschaltung zwischen ***„kleinem"*** und ***„großem" Datenmodell*** hat sich eher zufällig in den Linker-Dialog verirrt, da dies eigentlich eher etwas mit der Codeerzeugung und folglich mit dem Compiler zu tun hat.

Zuerst einmal die Frage: Was hat es eigentlich mit diesen Datenmodellen auf sich?

Wenn im vom Compiler generierten Code auf globale, statische Daten zugegriffen werden soll, geschieht dies normalerweise über absolute Adressen. Die Daten selbst können dann irgendwo im Speicher liegen und über beliebig viele Daten-"Hunks" verteilt sein. Auch die Gesamtgröße dieser Daten ist nur durch die Speichergröße des Rechners begrenzt. Diese kanonische Art der Adressierung bezeichnet man in diesem Zusammenhang als „großes" Datenmodell („Large Data").

Das klappt zwar recht gut und unterliegt auch keinerlei Beschränkungen, aber wie so oft gibt es auch hier Leute, die daran etwas zu mosern haben. Zum Beispiel benötigt eine absolute Adresse im

Code immer 32 Bit, also vier Bytes. Das bläht nicht nur die Codegröße auf, sondern kostet auch ein wenig mehr an Laufzeit. Ferner müssen alle diese Adressen korrigiert werden, wenn das Programm in den Speicher geladen wird, was in der Dateigröße einen gewissen Overhead bedeutet.

Deshalb ist ein anderer Ansatz, das sogenannte „kleine" Datenmodell, in gewisser Hinsicht schlauer. Hier wird ein Prozessorregister, nämlich das Adressregister A4, als Zeiger auf die statischen Daten des Programms benutzt. Die Daten werden jetzt mit einem 16 Bit-Offset relativ zu diesem Datenregister adressiert, was jeweils zwei Bytes Code und ein Nichtigstel an Laufzeit spart. Aber jetzt kommen die Nachteile von „Small Data":

- Mit 16 Bit kann man nur 65536 Bytes adressieren - also ist dies die Obergrenze für die globalen Daten des Programms.
- Alle Datenhunks des Programms müssen in einem einzigen Small-Data-Hunk zusammengefaßt werden. Insbesondere unterscheidet man dann nicht mehr zwischen initialisierten Daten („DATA"-Hunk) und den platzsparend abgespeicherten uninitialisierten Daten („BSS"-Hunk).
- Bei der Codegenerierung kann das Register A4 nicht für Variablen oder Zwischenergebnisse genutzt werden.

Welche Konsequenz soll man aus diesen Vor- und Nachteilen ziehen?

So ganz allgemein wohl gar keine. Im Zweifelsfall sollten Sie einfach ausprobieren, mit welchem Datenmodell ihr Programm sich besser macht.

Rein praktisch wird das Datenmodell in *MCPP* mit dem entsprechenden Gadget im „Linker"-Dialog und beim „mcppc" mit der Option `"-gd"` bzw. `"-gD"` eingestellt.

Noch eine Anmerkung zum Schluß: Ich erwähnte oben, daß MaxonC++ bei „large data" das Register A4 munter benutzt. Wie paßt das mit der Aussage zusammen, daß man small- und large-data-Code beliebig mischen kann? Nun, A4 wird auch bei „großem" Datenmodell nur in solchen Funktionen getrasht, die selbst keine weiteren Funktionen aufrufen - und folglich insbesondere keine Funktionen mit A4-relativen Daten. Natürlich wird in solchen Funktionen A4 auch gegebenenfalls auf den Stack gerettet und am Ende der Funktion wieder hergestellt. Damit merkt die gesamte „Umwelt" nichts davon, wenn in einer large-data-Funktion A4 mißbraucht wird.

2.5.3.2 Linken oder nicht Linken - das ist nier die Frage

Auch unter MaxonC++ haben Sie die Möglichkeit, d[illegible]inken ganz oder teilweise zu unterlassen, wenn auch nicht mit ganz so vielen Optionen wi[illegible] mcppc. Dazu dient das Gadget ***„Linkermodus"*** im Linker-Requester. Die vier Einstellungen:

- ***„Komplett linken":*** Das übersetzte Programm wird ganz normal mit den Bibliotheksroutinen zu einem ausführbaren Programm zusammengebunden - oder zumindest wird es versucht.
- ***„Nicht linken":*** Es werden keinerlei Bibliotheks- oder sonstige Module hinzugeladen. Der Linker meldet auch keine Fehler, sondern wird überhaupt nicht erst gestartet. Dieser Modus ist

recht angenehm, wenn man an einem Executable nicht interessiert ist und lediglich eine Objektdatei haben möchte.

☞ ***„Ohne Startup-Code“:*** Vorsicht, eine Einstellung für Spezialisten - wenn man damit nicht umgehen kann, stürzen die erzeugten Programme gnadenlos ab! Näheres über diese Option dazu finden Sie im Linker-Kapitel, und zwar in Abschnitt 4.6.

☞ ***„Library erzeugen“:*** Bei dieser Einstellung wird nach Möglichkeit versucht, das Programm zu linken. Vor allem wird aber während der Codeerzeugung für jede Funktion und jede globale Variable ein eigener „Hunk“ erzeugt (wenn Sie nicht wissen, was ein Hunk ist, macht das nix. Sie können aber auch einfach im Index nachschlagen!). Die Folge: Der Linker kann nicht benutzte Funktionen einfach herausschmeißen und so die Codegröße verringern. Dies ist praktisch, wenn man eine Bibliothek programmieren möchte, kostet ansonsten aber ein bißchen was: Funktionen einer Übersetzungseinheit können z. B. identische Zeichenketten nicht mehr gemeinsam benutzen und sich vor allem gegenseitig nicht mehr PC-relativ aufrufen.

2.5.3.3 Verhalten bei Fehler

Die 1.x-Version von *MCPP* versuchte immer, das Programm zu linken, auch dann, wenn bei der Übersetzung Fehler entdeckt wurden. Mit dem Schalter ***„Bei Fehler“*** können Sie dieses vielerseits unbeliebte Verhalten unterbinden: Wird bei der Übersetzung ein Fehler (gilt nicht für Warnungen!) gefunden, wird der Linkvorgang schlicht unterlassen.

2.5.3.4 Bibliotheken

Der Linker von MaxonC++ muß natürlich auch irgendwie wissen, woher er die Standard-Bibliotheken laden muß. Deshalb trägt man die entsprechenden Pfade im ***„Lib-Pfade“***-Gadget des Linker-Dialogs ein. Genau wie bei den Includedateien (siehe 2.5.2.1) kann man auch hier bis zu zwei verschiedene Pfade angeben und sich Dateien dabei automatisch umkopieren lassen. Von sich aus versucht der Linker immer nur, undefinierte Symbole im *„Logfile“* zu suchen und anschließend die entsprechenden Bibliotheksmodule zu laden. Wenn Sie eine Linker-Library, etwa die allseits beliebte *„amiga.lib“*, einbinden wollen, müssen Sie deren Namen (ohne *„.lib“*) im Gadget ***„Link-Libs“*** eintragen. Da es nur ein solches Gadget gibt, können Sie darin auch mehrere solcher Bibliotheken durch Leerzeichen getrennt eintragen, z. B. `"amiga debug murx"`. Diese Bibliotheksnamen können wahlweise mit einem vollständigen Pfad oder ganz ohne Pfad angegeben werden. In letzterem Fall sucht der Linker sie wie die „normalen“ Bibliotheksmodule im dafür spezifizierten Verzeichnis.

2.5.3.5 Zusammenfassen von Hunks

Jedes Modul eines Programms besitzt normalerweise einen Code-Hunk und außerdem noch bis zu drei verschiedenen Datenhunks. Bei einem Programm, das aus mehreren Übersetzungseinheiten besteht, kommt da schon so einiges zusammen, einmal ganz zu schweigen von den Bibliotheksmodulen. Damit das mit der Hunkanzahl nicht Überhand nimmt, ist es üblich, gleichartige Hunks zu einem einzigen zusammenzufassen.

Je weniger Hunks ein Programm besitzt, desto kleiner ist die ausführbare Programmdatei, denn jeder Hunk benötigt einen Overhead von ein paar Bytes. Andererseits ist es aber auch nicht schlau,

grundsätzlich alle jeweils gleichartigen Hunks zusammenzufassen, denn ein 250 kByte großer Hunk benötigt nun einmal einen ebenso großen, zusammenhängenden Speicherbereich. Bei Rechnern, die knapp an RAM sind, kommt es durchaus vor, daß der Speicher so fragmentiert ist, daß zwar durchaus noch fünf Blöcke zu je 50 kByte, aber kein einziger zusammenhängender 250k-Block frei ist - ein Programm mit einem unangemessen großen Hunk könnte also nicht mehr geladen werden. Deshalb kann man eine Höchstgrenze wählen, die der Linker beim Kombinieren von Hunks nicht überschreiten soll. Der entsprechende Betrag ist (in kByte) im entsprechenden Gadget des Linker-Requesters einzutragen. Wenn Sie z. B. aus irgendwelchen Gründen wollen, daß überhaupt keine Hunks zusammengefaßt werden, setzen Sie den Wert einfach auf Null.

In einem Fall nimmt sich der Linker aber auf jeden Fall eine gewisse Freiheit heraus: Bibliotheksmodule werden immer zu Blöcken von 10 kByte kombiniert, auch wenn Sie einen kleineren Wert vorgegeben haben. Das liegt daran, daß die automatisch eingebundenen Bibliotheksfunktionen naturgemäß aus etlichen kleinen Hunks bestehen und es einfach Wahnsinn ist, jeden davon im ausführbaren Programm einzeln auftauchen zu lassen.

2.5.3.6 Schreiben von Dateien

Einer der Vorzüge einer integrierten Umgebung wie *MCPP* besteht darin, daß sich so ziemlich alles im RAM und ohne Disketten- bzw. Festplattenzugriffe abspielen kann: Die Quelltexte werden aus dem RAM gelesen und das ausführbare Programm ebenda abgelegt.

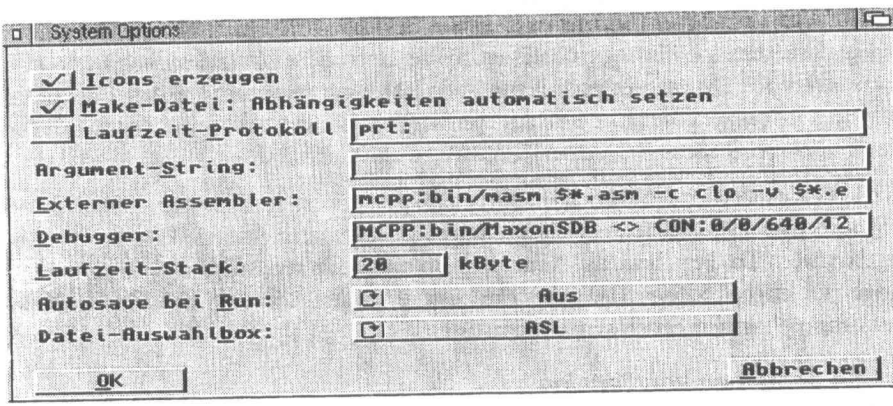
Nun gibt es aber bestimmte Gründe dafür, sich stets eine aktuelle ausführbare Programmdatei schreiben zu lassen.

Deshalb gibt es im „Linker"-Dialog ein Symbol, das Sie aktivieren können, wenn nach jeder erfolgreichen Übersetzung eine Exe-Datei (natürlich einschließlich der zugehörigen *„.mdbg"-Datei*, falls die entsprechende Compileroption angewählt ist) automatisch erzeugt werden soll.

Mit einem weiteren Gadget können Sie sich außerdem Objektdateien automatisch erzeugen lassen. Im Make-Modus geschieht das natürlich sowieso, so daß dieses Gadget dann keine Bedeutung hat.

2.5.4 MCPP-Systemeinstellungen

Mit dem dritten Dialogfenster, das Sie mit **„System"** aus dem **„Optionen"-Menü** öffnen, können diverse Einstellungen der MCPP-Umgebung vorgenommen werden.



2.5.4.1 Erzeugen von Icons

„Icons erzeugen" legt fest, ob MaxonC++ beim Schreiben von ausführbaren Dateien, Makefiles und Debugger-Dateien jeweils ein Icon erzeugen soll.

2.5.4.2 Abhängigkeiten in Make-Dateien

Die integrierte Projektverwaltung entscheidet souverän, wann welches Modul eines größeren Programms neu compiliert werden muß. Dazu muß sie aber wissen, von welchen Quelltextdateien ein Objekt abhängig ist. Diese Abhängigkeiten kann der Benutzer durchaus von Hand setzen - bequemer und sicherer ist es allerdings, wenn man diese Aufgabe MCPP überläßt.

Wenn sie das Symbol *„Make-Datei: Abhängigkeiten automatisch setzen"* angewählt haben, paßt *MCPP* beim Compilieren auf, welche Includedateien bei den einzelnen Objektdateien gelesen werden, und trägt diese dann als Abhängigkeit in die Make-Datei ein.

2.5.4.3 Protokollieren von Ein- und Ausgaben

Wenn Sie die Option *„Protokoll"* einschalten, werden sowohl die Ausgaben des Programms als auch die Eingaben des Benutzers in eine Datei geschrieben. Der Dateiname ist im Textgadget einzutragen, wobei `"prt:"`, also der Drucker, als Defaultwert vorgegeben wird.

2.5.4.4 Argumente aus der imaginären Kommandozeile

Beim Start von der Shell kann man Programmen Argumente übergeben, die in C mit den Parametern `"argc"` und `"argv"` ausgewertet werden können. Auch beim Programmstart aus der Entwicklungsumgebung von MaxonC++ kann man diese Art der Parameterübergabe simulieren, indem man sie im Textgadget `"Argumente"` eingibt. Diese Argumentzeile wird dann an aus *MCPP* gestartete Programme übergeben. Beachten Sie bitte, daß der erste Teil des Strings als Programmname interpretiert und in `"argv[0]"` übergeben wird.

2.5.4.5 Einbindung des Maxon Assemblers

Seit Version 3 beherrscht das integrierte Make auch sogenannte Regeln, die möglichenfalls automatisch angewendet werden, wenn für eine Objektdatei keine Compileranweisung vorhanden ist. Zwei dieser Regeln sind sogar schon „eingebaut": Zum einen die `".c.o"`-Regel, die besagt, daß eine auf `".o"` endende Datei, zu der eine `".c"`-Datei gleichen Namens existiert, als C-Programm compiliert werden kann, und zum anderen die `".asm.o"`-Regel für Assemblersourcen. Diese Regel können Sie über das Eingabefeld ***„Externer Assembler"*** frei bestimmen.

Default ist

```
mcpp:bin/masm $*.asm -c cl -v $*.err
```

Das `"$*"` wird (wie immer bei Make-Regeln) durch den endungslosen Dateinamen ersetzt. Also wird für jede auf `".o"` endende Objektdatei, für die es im Makefile keine Compileranweisung, dafür aber auf der Platte eine passende `".asm"`-Datei gibt, automatisch ein solcher Maxon Assembler-Aufruf erzeugt.

Diese Regel wird auch benutzt, wenn mit dem Menüpunt **„Text und Objekt aufnehmen"** eine auf `".asm"` endende Quelltextdatei in das Makefile aufgenommen wird. Auch in diesem Fall erhalten Sie also automatisch einen passenden Assembleraufruf.

Vielleicht fällt Ihnen die `"-v"`-Option auf. Der Maxon Assembler ist inzwischen so weit in das System integriert, daß *MCPP* die MaxonAsm-Fehlerdateien interpretieren kann (und genau diese werden durch `"-v"` erzeugt). Wenn MCPP ein Assemblermodul auf diese weise „extern" übersetzen läßt, liest es anschließend diese Fehlerdatei ein und zeigt die Assemblerfehler im normalen Compiler-Fenster an. Prima, nicht?

2.5.4.6 Debugger

Der Menüpunkt **„Debugger"** ruft bekanntlich den MaxonSDB auf (siehe auch 2.4.3). Damit *MCPP* auch weiß, was dabei zu tun ist, gibt es das Gadget ***„Debugger"*** im System-Einstellfenster. Hier ist die Shell-Anweisung anzugeben, die von *MCPP* ausgeführt werden soll (bis auf den Dateinamen, denn den hängt *MCPP* einschließlich des Argumentstrings immer automatisch an). Default ist

```
mcpp:bin/MaxonSDB <> CON:0/0/640/120/MaxonSDB
```

Wie Sie sehen, wird hier gleichzeitig das Fenster für die Ein- und Ausgaben des debuggten Programms festgelegt.

2.5.4.7 Stackgröße

Wenn ein Programm aus *MCPP* gestartet wird, braucht es natürlich auch einen Stack, und im Zweifelsfall nicht zu wenig davon. Die gewünschte Größe kann ebenfalls im „System"-Dialog eingestellt werden.

2.5.4.8 Autosave

Wenn die ***„Autosave"***-Option vom *MCPP* eingeschaltet ist, werden beim Start eines Programms aus der integrierten Umgebung automatisch alle Editor-Texte abgespeichert, sofern dies noch nicht

geschehen ist - bestimmt eine sinnvolle Maßnahme, zumal man in C stets ohne Netz und doppelten Boden programmiert. Dieser Autosave sollte nicht mit dem zeitabhängigen Autosave des *Edward* verwechselt werden.

Es gibt hier sogar gleich drei verschiedene Einstellungen:

☞ ***Aus:*** Der MCPP-Autosave unterbleibt.

☞ ***Texte sichern:*** Die Texte im Editor werden nötigenfalls abgespeichert.

☞ ***Texte und Session:*** Zusätzlich werden die *„.SESSION"-Dateien* von *Edward* und *MCPP* geschrieben. Dadurch haben Sie nach einem Absturz und erneutem Start von *MCPP* wieder alles so vor sich, wie es gewesen war.

2.5.4.9 Dateiauswahlbox

Der RCT-Filerequester hat seine Fans (mich eingeschlossen), aber der Commodore sieht es gar nicht gern, wenn man gegen seine Richtlinien verstößt. Deshalb kann zwischen ***RCT-*** und ***ASL-Dateiauswahlbox*** frei gewählt werden.

2.5.5 Abspeichern der Konfigurationsdatei

2.5.5.1 Allgemeines

Wenn im aktuellen Verzeichnis keine Datei *„mcpp.SESSION"* gefunden wird, liest *MCPP* die Einstellungen aus der Datei *„s:mcpp.CONFIG"* (siehe auch Abschnitt 1.3.3), und wenn es die auch nicht gibt, werden eben die internen Defaults verwendet. Mit dem Menüpunkt **„Optionen sichern"** werden alle aktuellen Einstellungen von *MCPP* (ausgenommen natürlich der Editor-Zustand) in *„s:mcpp.CONFIG"* abgespeichert.

An dieser Stelle soll noch darauf eingegangen werden, wie diese Konfigurations- und Session-Dateien eigentlich aufgebaut sind. Zwar werden Sie darin kaum einmal etwas von Hand ändern, aber Sie müssen es wissen, wenn Sie über den ARexx-Befehl `"setopt"` MCPP-Einstellungen ändern (siehe 2.6.3.6)

Die Konfigurationsdateien sind Texte mit einer beliebigen Folge von Optionen, die jeweils mit einem `"-"` beginnen. Dabei entsprechen die Compiler- und Linker-Einstellungen jeweils exakt denen von „mcppc": `"-gO"` schaltet die Optimierung ein, `"-wA"` warnt bei verdächtigen Zuweisungen in Bedingungen usw., etc. und so fort.

Für alle anderen Einstellungen von *MCPP* gibt es spezielle Optionen, die alle mit `"-x"` beginnen und vom *mcppc* naturgemäß nicht verstanden werden.

2.5.5.2 Der Editor

Zunächst ist da ein Eintrag `-x"Edward:Edward"`

Wie Sie sich vielleicht denken können, entnimmt *MCPP* dieser Angabe, woher der Editor *Edward* geladen werden soll. Falls Edward in Ihrem System nicht unter diesem Pfad und Namen zu finden

sein sollte, können Sie diesen Eintrag ändern. Sie sollten aber nicht unbedingt versuchen, auf diese Weise einen anderen Editor als *Edward* einzubinden, denn nach derzeitigem Stand beherrscht kein fremder Editor das kompilzierte Message-Protokoll zwischen *MCPP* und *Edward*. Alternative Editoren können ausschließlich über ARexx (Kapitel 2.6) integriert werden.

2.5.5.3 Fensterdaten

Am Anfang der Konfigurationsdatei finden Sie gewöhnlich eine ganze Reihe von Einträgen, die mit `"-Xn"` beginnen (wobei `n` eine Zahl zwischen `1` und `9` ist). Diese Daten legen Position und Größe der Windows von MCPP fest und sollten nie manuell verändert werden.

2.5.5.4 Switches und Schalter

Die Optionen `"-X10"` bis `"-X18"` repräsentieren diverse Schalterstellungen der integrierten MCPP-Umgebung. Es folgen jeweils stets ein Komma und die Schalterstellung - meist `"0"` oder `"1"`, manchmal aber auch höhere Nummern.

Und das bedeuten diese Einträge:

`-X10:` Exe-Datei automatisch erzeugen?
`-X11:` Objektdatei automatisch erzeugen?
`-X12:` Bei Compilerfehlern Linken unterlassen?
`-X13:` Icons erzeugen?
`-X14:` Protokolldatei schreiben?
`-X15:` Autosave (0 = Aus, 1 = Texte sichern, 2 = Texte + Session)
`-X16:` Größe des Laufzeitstacks
`-X17:` Linkermodus (0 = Nicht linken, 1 = Komplett linken, 2 = Ohne Startupcode, 3 = Library erzeugen)
`-X18:` Dateiauswahlbox (0 = RCT, 1 = ASL)

2.5.5.5 Stringoptionen

Die Einträge `"-X20"` bis `"-X23"` geben den Inhalt von vier Texteingabefeldern wieder und erwarten folgerichtig als Argument eine `"so"` eingeschlossene Zeichenkette:

`-X20:` Name der Protokolldatei
`-X21:` Argumentstring bei Programmstart aus der Umgebung
`-X22:` Default-Assembleraufruf
`-X23:` Debuggeraufruf

2.6 Die ARexx-Schnittstelle

2.6.1 Grundlagen

Die Programmiersprache Rexx ist in ihrer Amiga-Portierung „ARexx" zu beachtlicher Popularität gelangt, obwohl Rexx im Wirklichkeit doch eigentlich eher eine Script-Sprache und für „echte" Programme ziemlich unbrauchbar ist. Der Grund für diesen phänomenalen Erfolg ist (einmal abgesehen davon, daß ARexx ab Workbench 2.0 zum Lieferumfang jedes Amigas gehört), daß ARexx sich in hervorragender Weise dazu eignet, Programme „von außen" zu steuern und so zum einen komplexe Abläufe zu automatisieren und zum anderen wildfremde Programme zu einem System zu kombinieren.

Voraussetzung dafür ist, daß die Programme einen sogenannten ARexx-Port besitzen. Ein ARexx-Programm kann Anweisungen an eine solche Software-Schnittstelle schicken, und wenn das entsprechende Programm über einen geeigneten Befehlssatz verfügt, kann der Rexx-Programmierer Programmfunktionen mit einem ARexx-Script fernsteuern.

Der Editor *„Edward"* besitzt eine solche Schnittstelle, und da *Edward* Anweisungen an *MCPP* weitergeben kann, läßt sich die Compiler-Umgebung natürlich über diesen Umweg steuern.

Ein Beispiel: Das ARexx-Script

```
ADDRESS "EDWARD" 'SendAppComm "compile"'
```

veranlaßt *MCPP*, den im Editor befindlichen Quelltext zu übersetzen. Voraussetzung dafür ist natürlich, daß MaxonC++ zuvor gestartet wurde. Mit der ARexx-Anweisung `"ADDRESS"` legt man übrigens fest, wie der Port heißt, mit dem das Script kommunizieren soll.

Es bleibt noch die Frage offen, was man macht, wenn man *MCPP* ohne *Edward* benutzen und fernsteuern möchte - z. B. weil man einen anderen Editor bevorzugt (nahezu völlig unverständlich!) und diesen über ARexx in das MaxonC++ System integrieren möchte.

2.6.2 Der ARexx-Port

Seit Version 3 besitzt MCPP einen eigenen ARexx-Port. In Abschnitt 2.1 wurde bereits angedeutet, daß *MCPP* dafür vom CLI bzw. der Shell aus mit der Option `"-r"` zu starten ist, z. B. `MCPP -r`

In diesem Fall wird ein Port mit dem Namen `"MCPP"` eingerichtet. Falls ein solcher schon existiert (kommt insbesondere vor, wenn MaxonC++ mehrfach gestartet wird), werden so lange `"MCPP1"`, `"MCPP2"` usw. ausprobiert, bis ein noch nicht belegter Name gefunden wird. Alternativ können Sie auch hinter dem `"-r"` einen beliebigen Portnamen vorgeben, z. B.

```
MCPP -r "Bla Fasel Sülz!"
```

Generell werden solche Portnamen in Großbuchstaben umgewandelt. Auch hier werden systemweit eindeutige Namen sichergestellt und ggf. `"BLA FASEL SÜLZ!1"` usw. ausprobiert.

In jeden Fall startet nun *MCPP*, meldet sich und gibt zur Kontrolle den Portnamen aus. *Edward* wird nicht geladen, *MCPP* kommt im Rexx-Modus völlig ohne ihn aus.

Wenn Sie den Default-Namen verwenden, können Sie Ihr Script also locker-flockig mit `ADDRESS "MCPP"` einleiten. Welche Anweisungen Ihnen dann zur Verfügung stehen, wird in den folgenden Abschnitten dargestellt. Zuvor sollten Sie aber noch erfahren, wie Sie das MaxonC^{++} wieder loswerden, ohne Ihren Rechner booten zu müssen. Zum einen gibt es dafür die Anweisung `"QUIT"`. Das folgende kurze ARexx-Script entfernt also *MCPP* aus dem System:

```
Address "MCPP" quit
```

An dieser Stelle sehen Sie auch, daß Groß-/Kleinschreibung von Befehlen in ARexx generell egal ist.

Andererseits reagiert *MCPP* im Rexx-Modus auch auf die üblichen Abbruchsignale. Wenn Sie *MCPP* aus der Shell ohne `"run"` gestartet haben, genügt also die Tastenkombination <Ctrl>+<C>. Falls Sie `"run"` verwendet haben, können Sie mit der Shell-Anweiung `"status"` die Prozessnummer herausfinden und diesen Prozeß dann mit der `"break"`-Anweisung abmurxen.

2.6.3 Die ARexx-Befehle

2.6.3.1 Schließen, öffnen und Beenden

Wie bereits in 2.6.2 erläutert, kann *MCPP* durch die Anweisung `quit` beendet werden. `shutdown` schließt alle Fenster, die *MCPP* gerade geöffnet hat, z. B. weil das ganze System Iconifizert werden soll. Anschließend können mit `reopen` alle zuvor geöffneten Fenster erneut geöffnet werden.

2.6.3.2 Übersetzen, Starten und Ähnliches

Die ARexx-Befehle von *MCPP* entsprechen im wesentlichen den Anweisungen, mit denen normalerweise *Edward* den Compiler steuert (also über `"SendAppComm"`) und die folglich auch in der *„.DEF"*-Datei verwendet werden. Allerdings haben einige Anweisungen hier Argumente, die zum Teil aber optional sind. So auch bei der wichtigsten Anweisung überhaupt: `"compile"`. `"compile hello.c"` veranlaßt *MCPP*, das gewohnte Compiler-Fenster zu öffnen und die Quelldatei *„hello.c"* zu übersetzen. ARexx-Neulinge sollten beachten, daß die Anführungszeichen in diesem Fall notwendig sind, damit ARexx `"compile"` und `"hello.c"` als Teile einer zusammengehörenden Anweisung versteht und vollständig an *MCPP* sendet.

`"compile"` ohne Dateinamen bewirkt die Compilation einer Datei namens *„default.c"* - falls vorhanden. Im Make-Modus (auch den gibt es natürlich im Rexx-Modus) werden die Dateinamen allerdings generell ignoriert und es wird stur nach der Make-Datei gearbeitet. Die Anweisung `run` startet ein übersetztes Programm, falls ein solches vorliegt. Im Gegensatz zur „normalen" Entwicklungsumgebung wird dabei nicht automatisch compiliert, und es wird erst recht nicht geprüft, ob eine erneute Übersetzung (z. B. wegen Veränderung von Quelltexten) nötig wäre.

Ein mit `run` gestartetes Programm kann mit der Anweisung `break` brutal abgebrochen werden. debug startet in gewohnter Weise den Debugger.

2.6.3.3 Requester und Informationen

Auch unter ARexx müssen Sie nicht darauf verzichten, die zahlreichen Optionen und Einstellmöglichkeiten des Systems über Dialogfenster zu wählen: **opsys** öffnet das System-Dialogfenster, **opcomp** öffnet das Dielogfenster für die Compiler-Optionen und **oplink** öffnet den Linker-Requester. Das Info-Fenster können Sie sich mit der Anweisung **info** ansehen, und nach einen Compilerdurchlauf läßt sich das bereits geschlossene Compiler-Fenster (in dem eventuell Fehlermeldungen stehen) mit der Anweisung **errors** erneut öffnen.

2.6.3.4 Make

Die Projektverwaltung steht auch im Rexx-Modus zur Verfügung. **"loadmk <Dateiname>"** lädt die angegebene Make-Datei. Wenn der Dateiname weggelassen wird, öffnet sich das Dateiauswahlfenster, und der Benutzer kann selbst eine Datei auswählen. Dagegen wird bei **newmk** stets eine vom Anwender zu benennende neue Projektdatei angelegt. Es ist nicht möglich, hier einen Dateinamen als Argument zu übergeben. **leavemk** vergißt die geladene Make-Datei und schaltet wieder in den Normal-Modus.

Zur Manipulation von Projekten stehen nur die beiden Funktionen zur Verfügung, die sich sinnvoll von einem Editor aus starten lassen: **"addtx <Quelltextname>"** fügt eine neue Quelltextdatei hinzu, und **"addtxobj <Quelltextname>"** trägt eine Quelltextdatei samt passender Objektdatei und geeigneter Compileranweiung in die Make-Datei ein. Dabei werden natürlich auch auf *„.asm"* endende Quelltexte sinnvoll behandelt.

Alle anderen Operationen kann der Anwender über das Make-Dialogfenster vornehmen. Es wird mit **openmake** geöffnet.

2.6.3.5 Schreiben von Dateien

Mit **"exefile <Dateiname>"** wird - falls ein übersetztes und gelinktes Programm vorliegt - eine ausführbare Programmdatei geschrieben, während **"objectfile <Dateiname>"** eine linkbare Objektdatei schreibt. Bei beiden Anweisungen ist der Dateiname optional. Wird er weggelassen, öffnet sich ein Dateiauswahldialog.

2.6.3.6 Optionen speichern und setzen

Die beiden Anweisungen **"saveconf <Dateiname>"** **"savesession <Dateiname>"** schreiben alle Einstellungen in die angegebene Datei. Der einzige Unterschied ist der Default-Name, den die Anweisungen verwenden, wenn kein Dateiname angegeben wird - dies sind nämlich *„s:mcpp.config"* und *„mcpp.SESSION"*. Systemeinstellungen können aus einem ARexx-Script aber auch beliebig gesetzt werden. Dazu dient die Anweisung **"setopt <Optionen>"** Als Argument ist eine beliebig lange Folge von Einstellungen erlaubt. Diese sind in dem Format anzugeben, in dem sie in der Konfig- und in den Session-Dateien stehen.

Beispiel:

```
"setopt -X21,"hello World!" -pp -b -X10,1"
```

setzt den Argumentstring für Programmstarts auf `"hello World!"` und schaltet als Default in den C-Modus, aktivert die Erzeugung von Debug-Dateien und läßt bei jeder Übersetzung automatisch eine Exe-Datei schreiben.

Die Compiler- und Linker-Einstellungen wie `"-pp"` oder `"-b"` sind identisch mit den Optionen des CLI-Compilers „mcppc". Alle Optionen, die die Entwicklungsumgebung betreffen, werden mit einer mit `"-x"` beginnenden Anweisung eingestellt und werden in Abschnitt 2.5.5 dieses Handbuchs erläutert.

2.6.3.7 Informationen über die Entwicklungsumgebung

Für elegante Lösungen sollten Scripts auch in der Lage sein, Informationen über den aktuellen Status von *MCPP* zu erhalten. `"getinfo MAKE"` liefert den Namen der momentan geladenen Make-Datei bzw. einen Leerstring. Mit `"getinfo STATE"` erhalten Sie einen String **„NONE"**, **„COMPILED"** oder **„LINKED"** und können so feststellen, ob ein übersetztes bzw. übersetztes und gelinktes Programm vorliegt. In beiden Fällen liegt er Ergebnisstring nachher in der Variablen `"RESULT"`. Es ist auch zu beachten, daß vorher mit der Rexx-Anweisung `Options results` die Datenübergabe von *MCPP* in das ARexx-Programm ermöglicht wurde.

2.6.3.8 Fehlermeldungen und Fehlerstellenansprung

Die Anweisung `"errorscript <dateiname>"` legt ein ARexx-Script fest, das von *MCPP* jedesmal aufgerufen wird, wenn der Compiler eine Fehlermeldung ausgibt. Diesem Script werden als Argumente Dateiname, Zeilen- und Spaltennummer sowie die eigentliche Fehlermeldung übergeben. So veranlaßt `"errorscript errors.rx"` *MCPP* bei jedem folgenden Compilerfehler zu einem Aufruf der Art:

```
rx errors.rx "hello.c" 8 1 "Syntax Error"
```

Ihr Script `"errors.rx"` kann dann mit den übergebenen Daten beliebiges tun, z. B. sie in einer Fehlerdatei abspeichern.

Ein `"errorscript"`-Aufruf ohne Argument schaltet diesen Aufrufmechanismus wieder ab.

Da die Fehlerdaten schlicht an das Argument von `"errorscript"` angehängt werden, können Sie trickreich auch weitere Argumente an das ARexx-Script übergeben:

```
"errorscript wagga.rx ERRORS argl! bläk"
```

bewirkt Aufrufe wie:

```
rx wagga.rx ERRORS argl! bläk "hello.c" ...
```

Analog funktioniert die Anweisung `"editscript <Scriptname>"`, die dafür gedacht ist, auch im Rexx-Modus Fehlerstellenansprung zu realisieren. Das angegebene Script wird aufgerufen, wenn der Benutzer im Compiler-Fenster nach Ansprung einer Fehlerposition verlangt, und erhält als Argumente Dateinamen, Zeilen- und Spaltennummer, z. B.:

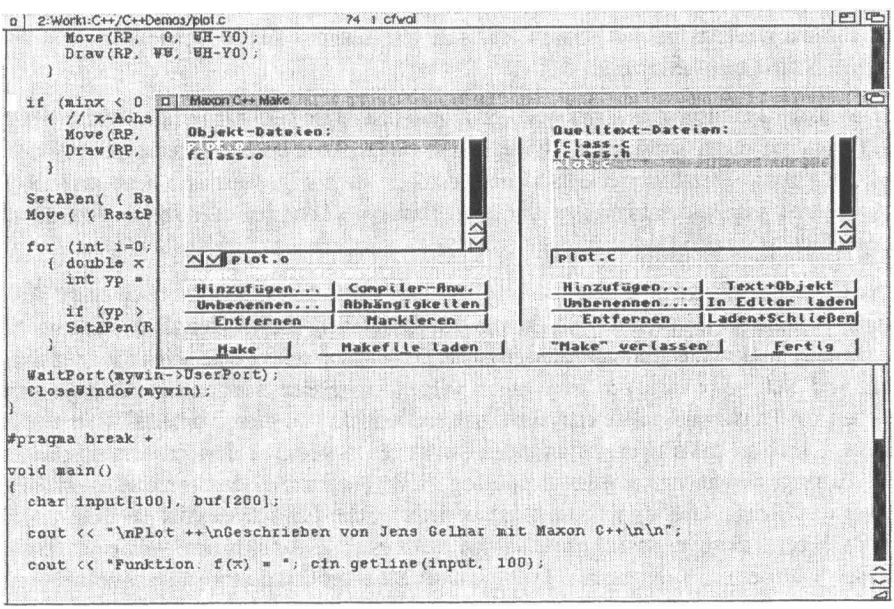
```
rx edit.rx "hello.c" 8 1
```

3. MAKE - eine integrierte Projektverwaltung

3.1 Das Prinzip

3.1.1 Wozu eine Projektverwaltung?

Größere Projekte bestehen normalerweise nicht aus einer einzelnen Quelltextdatei, sondern aus einer Vielzahl von Modulen, die vom Linker verbunden werden - zumindestens sollte es so sein. Das nimmt leicht Ausmaße an, die nicht mehr ganz leicht zu überschauen sind. Hinzu kommt, daß es natürlich Unsinn ist, nach jeder Änderung das ganze Projekt zu übersetzen. Auf die Dauer wird es aber lästig, sich immer wieder neu zu überlegen, welches Modul jeweils übersetzt werden muß, einmal ganz davon abgesehen, daß man eigenartige Überraschungen erleben kann, wenn man dabei etwas vergißt. So entsteht das Bedürfnis, solche Vorgänge zu automatisieren - wozu hat man schließlich einen Computer...



Die populärste Projektverwaltung, die auf so ziemlich allen ernstzunehmenden Rechnern anzutreffen und bei allen Compilersprachen tausendfach bewährt ist, heißt „make". Eigentlich ist „make" selbst eine Art Programmiersprache, und das macht es natürlich nicht leicht, ein komplettes Make in eine graphisch orientierte Entwicklungsumgebung wie die von MaxonC^{++} zu integrieren. Deshalb unterstützt *MCPP* nur einen recht begrenzten (aber den entscheidenden) Teil von „make". Übrigens wurde das oft kritisierte „make" von MaxonC^{++} 1.x vollständig überarbeitet und sollte jetzt kaum noch Wünsche offen lassen.

3.1.2 Was ist das überhaupt - ein Projekt?

Fangen wir doch einmal ganz graswurzelmäßig von vorn an: Woraus besteht ein Projekt (immerhin war von einer Projekt-Verwaltung die Rede) - und was ist überhaupt ein Projekt?

Also, zu den oben erwähnten Einschränkungen gehört, daß ein Projekt immer genau ein ausführbares Programm darstellt. Wenn man im Rahmen eines Projekts ein Programm in mehreren Versionen erstellen will (man denke da nur an „MCPP" und „mcppc", die im wesentlichen ein und das selbe Programm sind, nur mit unterschiedlichen aufgepfropften Benutzerschnittstellen), muß man eben für jedes einzelne Executable eine eigene Projektdatei erstellen.

OK - ein Projekt ist also ein ausführbares Programm. Und wie kommt man zu einem solchen Programm, sofern es sich nicht gerade um ein Bonsai-Miniatur-Projekt handelt?

Richtg, das erwähnte ich schon: Man linkt dafür Objektdateien, also Module, zusammen. Genaugenommen bindet der Linker auch noch fertige Bibliotheksmodule ein, aber das interessiert uns hier nur höchst periphär, oder genaugenommen nicht die Bohne. Der Freizeit-Existenzialist stellt sich jedenfalls nun die Frage „Und wo kommen die kleinen Module her?", der Coder weiß es schon: Das hat ausnahmsweise nichts mit Schweinskrams zu tun, sondern die Objektdateien werden von irgendwelchen Compilern erzeugt.

Zu jedem der Module gibt es genau einen ganz besonderen Quelltext, nämlich den Text, aus dem die Objektdatei durch Compilieren entsteht. Um so ein Objekt zu machen, braucht die Projektverwaltung also im wesentlichen eine Information darüber, wie der Quelltext dazu heißt, und außerdem noch ein paar Zusatzinformationen, etwa den Namen des Compilers oder irgendwelche der allseits beliebten Compiler-Optionen.

Aber das ist noch lange nicht alles, denn zu einemnormalen Projekt gehören noch einige Texte mehr - normalerweise Includedateien, die von den eigentlichen Quelltexten irgendwo und irgendwie benutzt werden. Das führt uns natürlich zu der Frage, wann und wie eine Objektdatei neu übersetzt wird. Nun, bevor wir alle in Rente gehen, werden wir vielleicht noch das Entwicklungssystem erleben, das Programme on-line compiliert, während man den Quelltext bearbeitet. Vorerst ist es jedenfalls noch so, daß man munter an seinen Quelltexten 'rumediert und irgendwann entscheidet, daß das Programm für eine erneute Übersetzung reif ist. Dann tritt die bereits mehrfach erwähnte Projektverwaltung „Make" in Aktion und schaut nach, welche Objektdateien für das Projekt nötig sind, und bevor diese zusammengelinkt werden, denkt es scharf darüber nach, welche der Objektdateien neu übersetzt („recompiliert") werden muß. Da ja der Compiler wenigstens zeitweise mit dem Programmierer Schritt halten soll, ist auch noch halbwegs einsichtig, wann das zu geschehen hat: Richtig, genau dann, wenn mindestens einer der Texte, aus denen das Modul entsteht (d. h. Quelltext und Include-Files), seit der letzten Übersetzung verändert wurde. Das läßt sich leicht an Erstellungsdatum und -uhrzeit der entsprechenden Dateien feststellen.

3.1.3 Was hat das ganze mit „Make" zu tun?

Die Projektverwaltung verwaltet im wesentlichen zweierlei Datenbestände: Texte und Objekte. Bei den Texten handelt es sich im allgemeinen um irgendwelche Quelltexte oder Includedateien, und aus diesen Texten erzeugt der Compiler unter der Kontrolle der Projektverwaltung die Objektdateien. Im letzten Schritt werden diese Objektdateien dann mit den Libraries zu einer ausführbaren Datei gelinkt.

Für jede der Objektdateien braucht das Make also Angaben darüber, wie es „gemacht" (daher der Name) werden soll, insbesondere den Namen eines Quelltexts und eines Compilers. Das würde man natürlich auch noch von Hand hinkriegen, und deshalb sagt man der Projektverwaltung auch, unter welchen Umständen ein Objekt compiliert werden soll und wann die alte Version noch einmal verwendet werden kann.

Wie bereits gesagt, muß ein Objekt neu übersetzt werden, wenn ein Text, der mit diesem Objekt zu tun hat, verändert wurde. Normalerweise handelt es sich bei diesen Texten um den Quelltext des Moduls sowie die darin benutzten Includedateien. Dazu vergleicht Make den Datestamp, also die Zeit der letzten Veränderung einer Datei, der Objektdatei mit denen „ihrer" Texte. Natürlich wird ein Objekt auch dann compiliert, wenn es bisher noch nicht existierte.

3.2 Die Praxis

3.2.1 Erstellen von Projekten

Alle Angaben über die Struktur eines Projekts werden sinnvollerweise in einer Datei festgehalten, dem Makefile. Mit MaxonC^{++} erstellen Sie diese Datei interaktiv mit dem Pull-Down-Menü **„Make"** und dem Make-Dialogfenster. Das Makefile wird dabei ständig aktualisiert, also nach jeder Veränderung neu abgespeichert. Da eine solche Datei normalerweise nicht sehr lang sein wird, dürfte dabei auch beim Diskettenbetrieb keine Verzögerung auftreten.

Am besten probieren Sie das Make-Utility einfach aus, indem Sie sich ein Mini-Projekt aus zwei Modülchen erstellen. Damit fangen Sie an, indem Sie den Menüpunkt **„Neues Makefile"** anwählen. Wenn bisher keine Projektdatei geladen ist, können Sie ebenso gut auch den Menüpunkt **„Projektverwaltung"** benutzen. Nun erscheint der Dateiauswahl-Dialog und fragt Sie nach dem gewünschten Dateinamen. Meist nennt man die Datei ganz einfach *„makefile"*, jedenfalls dann, wenn man nicht mehr als eine solche Datei in einem Verzeichnis hat. Wenn Sie dagegen an einem vorhandendem Projekt weiterarbeiten wollen, benutzen Sie den Menüpunkt **„Makefile laden"**.

Nach dem Erzeugen einer neuen Projektdatei öffnet sich ein Fenster, das zwei Scrollisten (eine mit ***„Objekt-Dateien"***, die andere mit ***„Quelltext-Dateien"*** betitelt) und eine ganze Menge Gadgets enthält. Klicken Sie hier am besten einfach auf das Gadget ***„Fertig"*** in der rechten unteren Ecke, um das Fenster wieder zu schlieflen.

An einem Projekt, daß mehr als einen Quelltext umfaßt, wird man in der Regel länger als einen Tag lang arbeiten. Deshalb sollten Sie die folgenden Tips beachten, um es dabei auf lange Sicht komfortabel zu haben:

- ☞ Zu jedem Projekt gehört ein eigenes Verzeichnis. Sie sollten also auf jeden Fall mit der Workbench oder Shell ein neues Unterverzeichnis in Ihrem *„MCPP:"*-Verzeichnis oder auch anderswo anlegen.
- ☞ Wenn Sie *MCPP* immer von der Shell starten, sollten Sie vorher stets in das Verzeichnis wechseln, in dem das *Makefile* liegt. Dadurch wird dies auch für MCPP und *Edward* zum aktuellen Verzeichnis, und dadurch werden die *SESSION-Dateien* von dort gelesen bzw. dahin geschrieben. Insbesondere werden dann auch die Compiler-Einstellungen von *MCPP* im passenden Verzeichnis abgelegt und beim nächsten Start von *MCPP* automatisch wieder eingeladen.
- ☞ *MCPP* kann von der Workbench durch Anklicken des Makefile-Icons gestartet werden. Die Projektdatei wird dabei automatisch als solche erkannt und geladen (und nicht etwa als Textdatei in den Editor). Dabei wird das Verzeichnis, in dem das Makefile liegt, zum aktuellen Verzeichnis von *MCPP* und *Edward*, was wiederum erfreuliche Auswirkungen auf die Behandlung der *SESSION-Dateien* hat (siehe oben).

Die bequemste Art, sich ein Projekt für MaxonC++ einzurichten, geht deshalb so:

1. Stellen Sie sicher, daß die Option ***„Icons erzeugen"*** eingeschaltet ist und daß ein geeignetes leeres Verzeichnis für Ihr Projekt existiert.
2. Erzeugen Sie in diesem Verzeichnis wie oben beschrieben eine Make-Datei, die Sie zunächst einmal so leer lassen, wie sie ist.
3. Beenden Sie jetzt *MCPP*.
4. Wenn Sie Workbench 2.0 (oder später) benutzen, machen Sie jetzt am besten ein **„Leave Out"** (**auslagern**) des Makefile-Icons. Dadurch liegt es immer sichtbar auf dem Workbench-Bildschirm und kann nach dem Einschalten des Rechners sofort geöffnet werden.
5. Starten Sie nun *MCPP*, indem Sie das Makefile-Icon (natürlich doppelt) anklicken. Nun werden *„Edward.SESSION"* und *„mcpp.SESSION"* automatisch aus dem Verzeichnis des Projekts gelesen (bzw. dorthin geschrieben).

Vielleicht erscheint Ihnen diese Prozedur etwas umständlich, aber auf die Dauer werden Sie es zu schätzen wissen, nach dem Booten des Rechners mit einem einfachen Doppelklick genau da weiterarbeiten zu können, wo Sie zuletzt aufgehört haben.

3.2.2 Hinzufügen von Dateien, umständliche Methode

Zu einem Projekt gehört natürlich mindestens eine Objektdatei, und um eine solche zu erzeugen, braucht man einen Quelltext. Versuchen wir es zunächst auf die umständliche Methode und schreiben einen Text, z. B.

```
#include <stream.h>
void test()  { cout << "Dies ist ein Test."; }
```

Diesen Text speichern Sie einmal als „*test.c*" ab (mit einer Datei ohne Namen kann Make nichts anfangen) und wählen dann den Menüpunkt **„Make/Projektverwaltung"**. Damit öffnen Sie wieder das Make-Fenster, das Sie schon beim Erzeugen der Projektdatei gesehen haben. Dabei ist Ihnen vielleicht auch schon aufgefallen, daß das Fenster vertikal zweigeteilt ist, wobei die linke Hälfte mit ***„Objekt-Dateien"*** und die Rechte mit ***„Quelltext-Dateien"*** überschrieben ist.Wenn Sie nun die Textdatei „*test.c*" und die Objektdatei „*test.o*" in Ihr Projekt aufnehmen möchten, geschieht dies eigentlich ganz intuitiv: Klicken Sie zunächst auf das Gadget ***„Hinzufügen..."*** in der rechten Spalte. Es öffnet sich nun ein Dateiauswahlfenster, in dem Sie die soeben erzeugte Datei „*test.c*" abwählen. Sie bestätigen die Auswahl, der Datei-Dialog wird geschlossen und schon steht „*test.c*" in der rechten Liste des Make-Fensters. Ebenso einfach wählen Sie jetzt das linke Gadget ***„Hinzufügen..."*** an und wählen in der Dateiauswahl den Dateinamen „*test.o*". Höchstwahrscheinlich existiert diese Datei noch überhaupt nicht, so daß Sie sie folglich nicht einfach anklicken können, sondern den Namen von Hand eintippen müssen.

Natürlich spielt es keine Rolle, ob sie zuerst - wie hier beschrieben - die Quelltextdatei und dann die Objektdatei oder umgekehrt eintragen. Die beiden Listen sind zunächst vollkommen voneinander unabhängig, und wenn Sie möchten, können Sie jetzt auf beiden Seiten beliebig viele Dateien eintragen, die allesamt auf der Platte vorhanden sein können, aber nicht müssen. Die Dateien müssen auch keineswegs im selben Verzeichnis wie die Make-Datei liegen, aber das ist allgemein üblich und wohl auch am praktischsten.

Wenn Sie jeweils mehrere Dateien aufnehmen, wird Ihnen auffallen, daß die Namen der Quelltext-Dateien alphabetisch sortiert werden, während die Objektdateien in der Reihenfolge erscheinen, in der Sie hinzugefügt werden. Dies ist auch die Reihenfolge, in der der Linker später die Objektdateien aneinanderhängt, was z. B. dann von Interesse ist, wenn man einen eigenen Startup-Code schreiben möchte. Die Reihenfolge der Objektdateien können Sie mit den beiden Pfeilgadgets links von der Anzeige für die angewählte Objektdatei verändern. Dumm wie Computer nun einmal sind, müssen Sie nun aber auch noch klarmachen, daß die Objektdatei „*test.o*" entsteht, indem „*test.c*" compiliert wird. Wählen Sie dazu „*test.o*" in der linken Spalte an (so daß der Name invertiert dargestellt wird und in der Anzeige unter der Objekt-Liste erscheint) und klicken Sie dann auf das Gadget ***„Compiler-Anw."***. Alternativ können Sie „*test.o*" auch mit einem Doppelklick anwählen. Nun öffnet sich ein neues Fenster, in dem alle Gadgets bis auf ein Cycle-Gadget oben links und die üblichen ***„OK"***- und ***„Abbruch"***-Gadgets am unteren Rand deaktivert sind. Das Cycle-Gagdet zeigt zunächst ***„Nicht übersetzbar"*** an. Nach einem Klick erscheint hier ***„intern compilieren"***, und alle Gadgets bis auf ein Texteingabefeld unten links werden gleichzeitig aktiv. Insbesondere kann jetzt auch die mit ***„Quelldatei"*** überschriebene Liste benutzt werden. Dort wählen Sie jetzt einfach „*test.c*" als Quelltext an und schließen das Fenster wieder mit einem Klick auf ***„OK"***.

Im Make-Fenster erscheint jetzt direkt unterhalb der Objekt-Liste „*mcppc -c test.c*". Dies ist nichts anderes als die Anweisung, mit der man in der Shell „*test.c*" nach „*test.o*" übersetzen würde.

3.2.3 Übersetzen des Projekts

Sie können das Programm jetzt compilieren, indem Sie entweder im Make-Fenster auf das Gadget *„Make"* klicken oder wie gewohnt im Editor die Taste <F9> bzw. das Menü **„Compiler/Übersetzen"** anwählen. Nun tritt Make in Aktion, stellt fest, daß *„test.o"* compiliert werden muß, und bindet anschließend aus diesem Objectfile ein Programm zusammen - oder versucht es zumindest, denn es gibt ja noch keine „main"-Funktion. Make funktioniert völlig unabhängig davon, welcher Text sich gerade im Editor befindet.

3.2.4 Hinzufügen von Dateien, einfache Methode

Beim folgenden Miniatur-Modul, das die noch fehlende „main"-Funktion enthält, machen wir es uns etwas einfacher:

```
#include "test.h"
void main()  {    test();  }
```

Diesen Text speichern wir einfach als *„main.c"* ab und wählen den Menüpunkt **„Text und Objekt hinzufügen"**. Jetzt macht Make vollautomatisch folgendes:

- ☞ Die Datei *„main.c"* wird in die Liste der Texte aufgenommen.
- ☞ Das Objekt *„main.o"* wird dem Projekt hinzugefügt.
- ☞ Die Compileranweisung für *„main.o"* wird passend gesetzt.

Jetzt fehlt dem Projekt nur noch eins: Die Header-Datei *„test.h"*, in der die Funktionen (na ja, hier ist es ja nur eine) des Moduls *„test"* deklariert. Die Datei enthält lediglich folgendes:

```
void test();
```

also den Prototypen der Funktion. So, wenn Sie jetzt den Compiler starten, müßte das Projekt korrekt übersetzt und gelinkt werden.

Zuvor sollten Sie aber *„test.h"* noch als Quelltext dem Projekt hinzufügen (im folgenden Abschnitt verrate ich Ihnen auch, wozu das gut sein soll). Das können Sie wie gehabt vom Make-Fenster aus erledigen. Wenn Sie die Datei aber sowieso gerade im Editor haben, können Sie auch den Menüpunkt **„Make/Text aus Editor aufnehmen"** benutzen, und schon wird der aktuelle Text des Editors in die Quelltext-Liste aufgenommen.

3.2.5 Abhängigkeiten

Das so ziemlich wichtigste Feature der Projektverwaltung haben wir aber bisher nicht benutzt: Die Abhängigkeiten. Offensichtlich benutzt in unserem Beispiel *„main.o"* die Datei *„test.h"* als Includedatei und ist folglich davon abhängig. Immer, wenn *„test.h"* verändert wurde, sollte deshalb auch *„main.o"* sicherheitshalber neu übersetzt werden.

Öffnen Sie also wieder das Make-Fenster, wählen Sie in der linken Liste die Objektdatei *„main.o"* an und klicken Sie dann auf das Gadget ***„Abhängigkeiten"***. Dabei werden Sie bemerken, daß dieses Gadget invertiert bleibt, bis Sie es erneut anklicken. Nun können Sie in der Quelltext-Liste alle die Dateien anwählen, von denen *„main.o"* abhängig sein soll, also insbesondere *„test.h"*. Korrekterweise können Sie auch *„main.c"* anwählen, aber bei interner Übersetzung weiß MaxonC^{++} von selbst, daß jede Objektdatei von ihrer Quelltextdatei anhängt. Jedenfalls werden Sie dabei merken, daß Sie jetzt in der rechten Spalte beliebig viele Zeilen gleichzeitig aktivieren und durch erneutes Anklicken wieder deaktiveren können. Dieser Modus bleibt erhalten, so lange das Gadget ***„Abhängigkeiten"*** invertiert ist. Sie können jetzt auch ganz einfach auf eine andere Objektdatei klicken, um deren Abhängigkeiten zu betrachten und zu manipulieren.

Im Grunde genommen ist dieses manuelle Setzen von Anhängigkeiten aber überhaupt nicht nötig. Es reicht völlig, wenn Sie im System-Dialogfenster (Menüpunkt **„Optionen/System"**) das vielversprechend mit ***„Make-Datei: Abhängigkeiten automatisch setzen"*** beschriftete Symbol anwählen. Wann immer nun eine Objektdatei des Projekts compiliert wird, merkt *MCPP* sich, welche Includedateien dabei gelesen wurden - natürlich nur die selbstgeschriebenen, die man mit `#include "Name"` einbindet, und nicht etwa die System-Includes, die mit `#include <Name>` includiert werden, denn jene haben in einer Make-Datei normalerweise nichts zu suchen. Anschließend werden die Abhängigkeiten der jeweiligen Objektdatei automatisch gesetzt, wobei nötigenfalls sogar neue Textdateien in das Projekt aufgenommen werden.

Deshalb beschränkt sich das Anlegen eines Projekts zusammengefaßt auf folgende Schritte:

- ☞ Erzeugen sie eine neue Projektdatei.
- ☞ Laden Sie alle „echten" Quelltextdateien, also keine Includefiles, einmal in den Editor und wählen Sie dann den Menüpunkt **„Text und Objekt aufnehmen"**.
- ☞ Starten Sie die Übersetzung, wobei Sie sicherstellen sollten, daß die Objektdateien allesamt noch nicht existieren. Dann werden selbsttätig alle Quelltexte durchcompiliert, die Includedateien in das Projekt aufgenommen und die Abhängigkeiten korrekt gesetzt.

3.2.6 Weitere Funktionen im Make-Fenster

3.2.6.1 Projektpflege

Es kommt hin und wieder vor, daß eine Quelltext- oder Objektdatei komplett überflüssig wird. Dann wählen Sie die betreffende Datei einfach auf der richtigen Seite des Make-Fensters an und klicken auf das Gadget ***„Entfernen"***. Nach einer Sicherheitsabfrage wird die Datei vollständig aus dem Projekt entfernt.

Ebenso kann es passieren, daß Ihnen ein Dateiname plötzlich nicht mehr ganz passend erscheint. Dies können Sie elegant erledigen, indem Sie den Dateinamen im Make-Fenster anklicken und dann das Gadget ***„Umbenennen..."*** benutzen (aber bitte das auf der richtigen Seite, denn auch diese Funktion gibt es gleichermaßen für Objekt- und Quelldateien). Es erscheint der übliche Datei-Dialog, dem Sie nun den gewünschten neuen Namen anvertrauen. Beachten Sie aber bitte, daß diese

Funktion die Datei ausschließlich in der Make-Datei umbenennt. Das eigentliche Umbenennen der Datei auf der Festplatte bzw. Diskette müssen Sie schon selbst erledigen.

3.2.6.2 Übersetzung erzwingen

Eigentlich entscheidet das Make ja selbständig und prinzipiell auch korrekt, welche Datei wann recompiliert werden soll. Manchmal möchte man dies aber auch erzwingen, z. B. wenn man Compileroptionen verändert hat und einige oder alle Objektdateien gezielt mit diesen neuen Optionen übersetzen lassen möchte. Deshalb gibt es im Make-Fenster das Gadget ***„Markieren"***.

Ähnlich wie ***„Abhängigkeiten"*** ist auch ***„Markieren"*** ein Umschaltbares Gadget. Während es aktiviert ist, können Sie in der Objektliste beliebig Dateien anwählen und deaktivieren. Beim nächsten Übersetzungsvorgang werden die so markierten Dateien dann auf jeden Fall compiliert, und zwar zusätzlich zu den Dateien, die Make ohnehin für übersetzungswürdig befindet.

3.2.6.3 Dateien in den Editor laden

Beim Programmieren hat man es fast immer nur mit den Textdateien zu tun, die zum aktuellen Projekt gehören. Deshalb bietet *MCPP* eine elegante Möglichkeit, Quelltexte in den Editor zu laden.

Öffnen Sie dazu das Make-Fenster und wählen Sie in der rechten Liste den gewünschten Quelltext an. Wenn Sie nun das Gadget ***„In Editor laden"*** anklicken, wird der Text - wer hätte das gedacht - in den *Edward* hinzugeladen. Das Gadget ***„Laden+Schließen"*** funktioniert genauso, schließt aber gleichzeitig das Make-Fenster. Dies könenn Sie auch mit einem Doppelklick auf den Dateinamen erreichen.

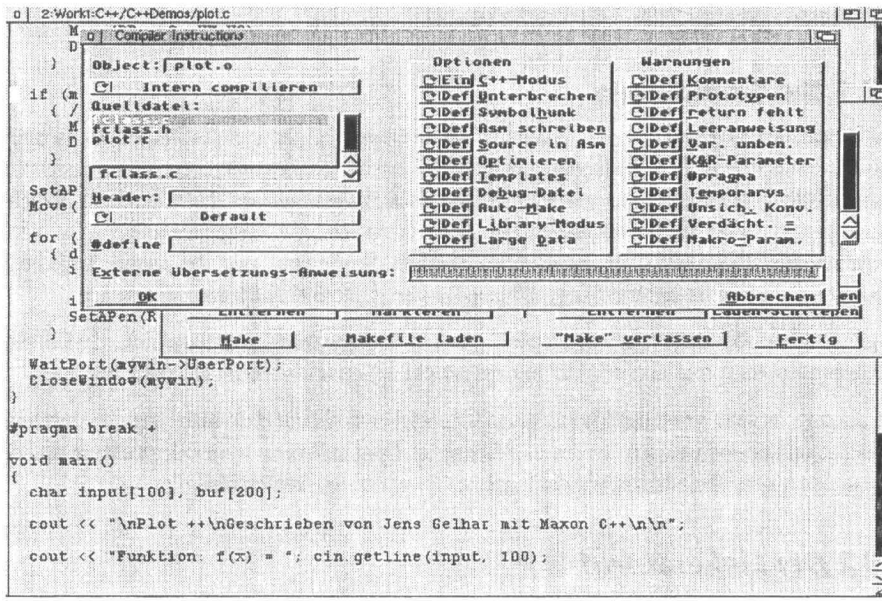
3.2.6.4 Projektdatei wechseln

Am unteren Rand des Make-Fensters befinden sich zwei bisher unerwähnte Gadgets: ***„Makefile laden"*** und ***„Make verlassen"***. Naheliegenderweise erscheint bei Anwahl des ersteren die Datei-Auswahlbox und ermöglicht Ihnen, eine andere Projektdatei zu laden. Mit dem anderen Gadget verlassen Sie den Make-Modus und gelangen wieder in die normale Betriebsart für Ein-Modul-Programme.

3.2.7 Compileranweisungen für Objektdateien

3.2.7.1 Interne Übersetzung

Zwischen dem Compileranweisungs-Requester der Projektverwaltung und dem globalen Compiler-Dialogfenster besteht auch optisch eine gewisse Ähnlichkeit. Betrachen Sie einmal die Gadgets, die benutzbar werden, wenn für eine Objektdatei der Modus ***„Intern compilieren"*** angewählt wurde. Sie werden feststellen, daß die wichtigsten Compileroptionen für jedes Modul des Projekts separat eingestellt werden können.



Allgemein gelten die globalen Einstellungen auch im Make-Modus, sofern eine Option nicht explizit anders eingestellt wird. Deshalb finden Sie im Compileranweisungs-Requester eine ganze Menge kleiner Cycle-Gadgets mit den Stellungen *„Def"*, *„Ein"* und *„Aus"*. Dabei steht *„Def"*, wie man sich denken kann, für *„Default"*, d. h. bei dieser Schalterstellung soll für das Modul die globale Einstellung gültig sein.

Bei den mit *„Optionen"* überschriebenen Gadgets entsprechen die ersten acht exakt den entsprechenden globalen Schaltern. *„Auto-Make"* entspricht einem Gadget aus dem „System"-Dialog und legt fest, ob die Abhängigkeiten der betreffenden Objektdatei vom Make automatisch gesetzt werden sollen, *„Library-Modus"* übersetzt ein Modul in handlichen kleinen Häppchen (siehe Einstellung *„Library erzeugen"* in Abschnitt 2.5.3.2), und *„Large Data"* entspricht der Einstellung *„Datenmodell: Groß"* im globalen Linker-Requester.

Die elf mit *„Warnungen"* überschriebenen Einsteller entsprechen natürlich exakt den elf globalen optionalen Warnings (siehe 2.5.2.5), und im *„#define"* überschriebenen Texteingabefeld kann optional ein zusätzliches Makro definiert werden. Es ist also nicht möglich, ein global definiertes Symbol für ein einzelnes Modul umzudefinieren.

Natürlich fehlen noch ein paar Compileroptionen, z. B. ist es über die Dialoge nicht möglich, für eine Übersetzungseinheit die Include-Pfade anders zu setzen. Solche (wohl eher ungewöhnliche) Einstellungen sind aber möglich, wenn Sie die Projektdatei in den Editor laden, „von Hand" verändern, abspeichern und dann erneut als Makedatei laden. In Abschnitt 3.3 erfahren Sie, wie die Projektdateien aufgebaut sind.

3.3 Handgemachte Makedateien

3.3.1 Die Grundlagen

Das Format der Projektdateien von MaxonC++ orientiert sich an den Standard-Makefiles, ist aber natürlich etwas strikter, da in der integrierten Entwicklungsumgebung ja nur ein gewisser Ausschnitt des bekannten Make implementiert werden konnte. Es ist dadurch relativ leicht, ein solches Makefile von Hand zu erstellen oder ein vorhandenes Makefile an MaxonC++ anzupassen. Wenn Sie nicht in der integrierten MCPP-Umgebung arbeiten, sondern die Option `"-m"` des mcppc benutzen, bleibt Ihnen sogar nichts anderes übrig, als Make-Dateien grundsätzlich manuell zu erstellen.

Es gibt zwei verschiedene Arten von Zeilen, nämlich Abhängigkeits- und Anweisungszzeilen. Sie unterschieden sich formal dadurch, daß letztere mit einem Leerzeichen oder Tabulator beginnen.

Ein „Eintrag" beginnt grundsätzlich mit einer Abhängigkeitszeile, die dem Make sagt, was erzeugt werden kann und von welchen Dateien es abhängig ist. Optional können eine oder mehrere Anweisungszeilen folgen, die festlegen, wie das fragliche Objekt erzeugt werden muß.

3.3.2 Der Linkeraufruf

MaxonC++ erwartet als ersten Eintrag stets die Exe-Datei, um die es geht (Sie erinnern sich: Leider immer nur ein Exefile pro Makefile). Die Exe-Datei ist - formal gesehen - von den Objektdateien abhängig, und erzeugt wird sie durch einen Linker-Aufruf. Als Linker muß hier der im *mcppc* benutzt werden (auch wenn Sie nicht *mcppc*, sondern *MCPP* verwenden). In diesem Zusammenhang sollte ich wohl erwähnen, daß es für das Verständnis dieses Kapitels enorm hilfreich wäre, wenn Sie schon ein wenig mit dem *mcppc* vertraut wären (siehe sonst Kapitel 5).

Eine Abhängigkeitszeile beginnt stets mit dem Namen der zu erzeugenden Datei, gefolgt von einem „:" und optional der Liste der Abhängigkeiten. Wenn unser Projekt „*Demo*" heißt und aus den beiden Modulen „*main.o*" und „*test.o*" besteht, beginnt unser Makefile also mit den Zeilen:

```
Demo: main.o test.o
 mcppc main.o test.o -o Demo
```

Dazu noch einige Anmerkungen:

☞ Aus Gründen der Kompatiblität zur ersten Version ist statt `"mcppc"` auch `"cppc"` erlaubt. Dieses Feature wird aber möglicherweise in späteren Versionen von MaxonC++ nicht mehr enthalten sein.

☞ Da das MaxonC++ Make keinen externen Linker aufruft, sondern alles implizit intern erledigt, ist auch die Option `"-o Demo"`, die den Namen der Exe-Datei festlegt, strenggenommen nicht erforderlich, zumal er aus der Abhängigkeitszeile bereits bekannt ist. Aus Ästhetischen Gründen sollte man diese Angabe aber trotzdem machen.

☞ MaxonC++-Make weiß, daß ein Linkeraufruf von den benutzten Objektdateien abhängig ist. Deshalb sind auch die beiden Einträge in der Abhängigkeitszeile nicht wirklich nötig.

3.3.3 Internes Compilieren

Nach dem Linkeraufruf folgen in beliebiger Reihenfolge die Einträge für die einzelnen Objektdateien. Wenn Ihnen danach ist, können Sie auch Einträge für Projektdateien hinzufügen, die überhaupt nicht zum Projekt gehören - die werden dann ignoriert.

Auch hier beginnt jeder Eintrag mit einer Abhängigkeitszeile, also z. B.

```
main.o: main.c test.h
```

Die Objektdatei *„main.o"* soll also bitteschön übersetzt werden, wenn eine der Dateien *„main.c"* und *„test.h"* in letzter Zeit verändert wurde. Fehlt noch der Compileraufruf: Der Compiler heißt ja bekanntlich genau wie der Linker *„mcppc"* (oder auch *„cppc"*, siehe oben). Die Option `"-c"` sorgt dafür, daß eine Objektdatei statt einer Exe-Datei erzeugt wird:

```
main.o: main.c test.h
  mcppc -c main.c
```

Außer `"-c"` kann auch jede andere Compiler-Option aus Kapitel 5 benutzt werden, z. B. `"-O"` zum Einschalten der Optimierungen. Bei allen Optionen, die nicht explizit gesetzt werden, werden die globalen Defaults verwendet.

Als Erbstück aus der 1.x-Version versteht MaxonC++ Make hier auch zwei alternative Compiler-Anweisungen:

`"cc"` ist gleichbedeutend mit `"mcppc -c -pc"` und

`"c++"` entspricht `"mcppc -c -pp"`. Dabei wird also explizit im ANSI C- bzw. im C++-Modus compiliert.

Es ist übrigens nicht zwingend erforderlich, dem Make zu sagen, daß ein Objekt von seiner Quelltextdatei (und das gesamte Projekt von seinen Objektdateien) abhängig ist, denn das weiß Make auch so. Also könnte man das Makefile folgendermaßen abkürzen:

```
Demo:  mcppc main.o test.o
test.o:
  mcppc -c test.c
main.o: test.h
  mcppc -c main.c
```

3.3.4 Externe Übersetzung

Jede Compileranweisung, die mit *„mcppc"* beginnt, wird von MaxonC++ „intern" ausgeführt, d. h. Make und Compiler sind Bestandteile eines einzigen Programms und keines von beiden muß

irgendwie nachgeladen werden. Leider ist MaxonC++ zugegebenermaßen nicht ausreichend (z. B. hat man öfters ein Assembler-Modul in seinem Projekt). Aber auch das ist kein Problem: Schreiben Sie die Anweisung ganz einfach in das Makefile! Sie können dann sogar mehrere nacheinender auszuführende Zeilen (=Shell-Anweisungen) für eine Objektdatei angeben, z. B. so:

```
sub.o: sub.asm
  mcpp:bin/masm sub.asm -c clos    echo "Assembler ist fertig"
```

Wenn Sie diese Projektdatei von der integrierten Umgebung MCPP ausführen lassen, wird vorher sogar geprüft, ob die Dateien, von denen das Objekt abhängig ist (in diesem Falle nur „*sub.asm*"), sich im Editor befinden und seit ihrer letzten Abspeicherung verändert wurden. Falls ja, werden sie automatisch abgespeichert, denn der externe Compiler/Assembler liest seinen Quelltext ja aus der Datei und nicht (wie MaxonC++) direkt aus dem *Edward*.

Speziell bei der Benutzung des Maxon Assemblers gibt es noch eine Besonderheit: Nach der Übersetzung schaut *MCPP* nach, ob eine zur Objektdatei passende „*.err*"-Datei erzeugt wurde (im Beispiel also „*sub.err*"). Falls ja, versucht *MCPP* diese als MaxonAsm-Fehlerdatei zu interpretieren und die Fehlermeldungen direkt im Compiler-Fenster anzuzeigen. Also wäre hier der Aufruf `masm sub.asm -c clos -v sub.err` höchst empfehlenwert.

Noch einmal zurück zu den Mehrfach-Anweisungen: Wie gibt man die eigentlich in MCPP ein, da es doch im Compileranweisungs-Dialog für externe Übersetzeraufrufe nur ein Textgadget gibt? Nun, hier sind die Zeilen mit einem Gegenschrägstrich `"\"` zu trennen. Davor und dahinter muß aus Gründen der Eindeutigkeit jeweils ein Leerzeichen stehen.

3.3.5 Weitere Features

Bisher wurden die Grundfunktionen von Make erläutert, die vom internen Make direkt und sichtbar unterstützt werden. Es gibt aber noch eine ganze Menge zusätzlicher Features, die besonders dann von Interesse sind, wenn man seine Projektdatei für den CLI-Compiler *mcppc* von Hand schreibt.

3.3.5.1 Zeilenverbindung

Die Anzahl der Dateien in einem Projekt wächst oft schnell an und mit ihr auch die Längen von Abhängigkeitszeilen oder Linkeraufrufen. Da z. B. Edward die Zeilenlängen auf 256 Zeichen beschränkt, stellt sich irgendwann das Problem, daß man mehr in eine Zeile packen möchte als eigentlich gesund ist.

Die Lösung ist die Zeilenverbindung mit dem Gegenschrägstrich. Endet eine Zeile mit `"\"`, so verbindet Make sie mit der nächsten Zeile. Auf diese Weise können beliebig viele „echte" Zeilen zu einer einzigen Make-Zeile zusammengefaßt werden.

3.3.5.2 Kommentare

Kommentare beginnen für Make mit einem `"#"`.

Beispiel:

```
# makefile für "Demo"
# Version vom 24.05.94
Demo: main.o  # heute nur eine Objektdatei
mcppc main.o -o Demo   # Linkeranweisung
```

Da Kommentare dem Make selbst ziemlich egal sind, werden sie auch nicht irgendwie gemerkt. Aber immerhin merkt sich MaxonC++ Make beim Lesen einer Projektdatei, ob darin Kommentare, Makros oder Regeln (siehe dazu die nächsten Abschnitte) vorkamen. Haben Sie als Benutzer von *MCPP* in der integrierten Umgebung an der Projekdatei etwas verändert, so daß das Makefile neu geschrieben werden muß, dann fragt *MCPP* sicherheitshalber noch einmal nach, ob das offensichtlich von Hand erstellte Makefile wirklich überschrieben werden soll. Es erscheint also eine Dialogbox mit dem Inhalt

Die Projektdatei „<name>“ wurde nicht
von MaxonC++ Make automatisch erstellt.
Wollen Sie sie trotzdem überschreiben?
Ja Nein

Sie haben dadurch bei Bedarf noch Zeit, z. B. eine Shell zu öffnen und Ihr mühsam handgeschriebenes *Makefile* vor der Vernichtung durch *MCPP* zu retten. Sie können die obige Nachfrage natürlich auch verneinen, aber dann wird das Update der Projektdatei nur aufgeschoben, nicht aufgehoben, und bei der nächsten Gelegenheit nervt *MCPP* erneut mit der selben Frage.

3.3.5.3 Makros

Auch einfache Makros können in Make verwendet werden. Dieses Feature geht aber keineswegs so weit wie z. B. die Makros im C-Preprozessor, vielmehr kann man im Text lediglich eine Zeichenfolge (den Makronamen) durch eine andere ersetzen lassen. Aber auch das ist hin und wieder ganz nützlich.

Man definiert sich ein Makro durch eine Zeile der Form

```
<Makroname> = <Inhalt>
```

Als Makroname ist dabei eine beliebige Folge von Buchstaben, Ziffern und dem Unterstrich `"_"` erlaubt.

```
Objects = main.o test.o sub.o
link_2 = mcppc -g40 -gsi -l amiga
X = Exefile
```

Nachdem ein Makro auf diese Weise definiert wurde, kann es benutzt werden. Dazu muß dort, wo das Makro durch seinen zugewiesenen Wert ersetzt werden soll, ein Dollarzeichen `"$"` vorangestellt werden. Ist der Makroname mehr als ein Zeichen lang, muß er außerdem auch noch in runden Klammern eingeschlossen werden. Nach den obigen Definitionen könnte eine Make-Datei also so beginnen:

```
$X: $(Objects)
$(link_2)
$(Objects)
```

Daran sehen Sie auch schon die beiden typischen Anwendungen von Makros im Makefiles: Zum einen spart man bei geeigneter Anwendung Tipparbeit, zum anderen muß man für bestimmte Änderungen wie das Hinzufügen weiterer Objektdateien nur noch an einer Stelle der Datei die Makrodefinition verändern, statt an mehreren Stellen den gleichen Mist einzutippen. Nebenbei räumen geschickt gewählte Makros den Quelltext auf und sorgen so für mehr Übersicht.

Es gibt einige vordefinierte Spezialmakros:

`"$$"` ist fest mit dem Dollarzeichen belegt. Wenn Sie an irgend einer Stelle Ihrer Projektdatei ein einzelnes `"$"` haben sollen, z. B. als Bestandteil eines eigentümlichen Dateinamens wie `"a$b"`, müssen Sie also ein doppeltes Dollarzeichen schreiben (`"a$$b"`), um Make davon abzuhalten, das nachfolgende (hier also `"b"`) als Makro zu behandeln. Außerdem gibt es noch `"$*"` und `"$@"`, die bei Regeln eine wichtige Rolle spielen:

3.3.5.4 Regeln

Es ist auf die Dauer ziemlich ermüdend, immer wieder von Hand einzutippen, daß Make aus einer auf *„.asm"* endenden Textdatei eine *„.o"*-Objektdatei erzeugen soll, indem es einem bestimmten Assembler mit bestimmten Optionen aufruft. Dieser Vorgang kann durch Make-Regeln erheblich abgekürzt werden.

Die Definition einer Regel sieht auf den ersten Blick wie ein ganz normaler Makefile-Eintrag aus, nur daß der Name der (Pseudo-)Zieldatei aus den Endungen von Quell- und Zieldatei zusammengesetzt ist und keine Abhängigkeiten angegeben werden dürfen. Eine typische Regel für Assembler-Module könnte z. B. so aussehen:

```
.asm.o:     masm -c clos $*.asm -o $@
```

Das Makro `"$*"` wird durch den seiner Endung beraubten Dateinamen, `"$@"` durch den vollständigen Namen der Objektdatei ersetzt.

MaxonC++ Make wendet Regeln nur an, um Objektdateien zu erzeugen, die in der Abhängigkeitszeile der Linkeranweisung genannt werden und für die es keinen Eintrag mit einer expliziten Compileranweisung gibt. Dann wird nach der ersten Regel gesucht, bei der die Endung der Zieldatei (meist `".o"`) stimmt und eine entsprechende Quelldatei auch existiert.

Beispiel:

```
.cc.o: mcppc -c -pp -O $*.cc
.c.o:  mcppc -c -pc -O $*.cc
Demo:  main.o test.o
   mcppc main.o test.o sub.o -o Demo
main.o:main.c test.h
```

Wenn nun die Dateien *„main.c"*, *„main.cc"*, *„test.cc"* und *„sub.cc"* existieren, wird für *„test.o"* die erste, für *„main.o"* die zweite und für *„sub.o"* überhaupt keine Regel benutzt - denn wenn *„sub.o"* nicht in der Abhängigkeitszeile der Exe-Datei auftaucht, sieht Make auch keinen Handlungsbedarf zur Erzeugung derselben.

Für *„main.o"* wurde im Beispiel sogar eine Abhängigkeitszeile eingeführt. Gibt es eine solche nicht, setzt Make automatisch den Namen der Quelldatei als einzige Abhängigkeit.

Es gibt beim MaxonC++ Make zwei eingebaute Regeln:

`.c.o: mcppc -c $*.c` übersetzt C- oder C++-Quelltexte. Beachten Sie, daß MaxonC++ Make bei interner Übersetzung als Compiler-Optionen die global eingestellten Defaults verwendet. Außerdem gibt es eine `".asm.o"`-Anweisung für Assembler-Module. In der integrierten Umgebung *MCPP* kann die zugehörige Anweisungszeile im System-Dialogfenster eingegeben werden (übrigens die selbe Anweisung, die auch benutzt wird, wenn der Menüpunkt **„Text und Objekt aufnehmen"** auf eine *„.asm"*-Datei angewendet wird), beim *mcppc* ist sie belegt mit: `.asm.o: masm $(AFLAGS) $*.asm`

Wenn Sie mit dem *mcppc* arbeiten, sollten Sie folglich dem Makro `"AFLAGS"` die passenden Optionen zuweisen, z. B. `"-c clo"`.

4. Der Linker

4.1 Prinzipielles

Ein Linker erfüllt im wesentlichen zwei Aufgaben:

☞ Er fügt Module eines Programms zusammen. Das ist ein nahezu unverzichtbares Feature, denn ab einer gewissen Größe ist es praktisch unmöglich, ein Programm in einem einzigen Modul zu schreiben.

☞ Anschließend bindet er vordefinierte Bibliotheksroutinen in das Programm ein. Auch das ist eine Aufgabe, die aus dem Programmieralltag nicht wegzudenken ist, denn jeder C- oder C++-Programmierer erwartet ganz selbstverständlich, daß simple Funktionen wie `"strcpy"` oder aufwendige Operationen wie `"printf"` in irgendwelchen Bibliotheken vorhanden sind und vom Linker anstandslos eingebunden werden.

Diese beiden Funktionen überschneiden sich allerdings, denn der Unterschied besteht einzig und allein darin, daß man dem Linker ausdrücklich sagt, welche Module er in ein Programm einbinden soll, während er die benutzten Bibliotheksfunktionen weitgehend selbständig hinzulinkt - und, wenn er hinreichend dumm ist, auch noch erheblich mehr - vor allem im Workstation-Bereich gibt es Compiler-/Linker-Systeme, bei denen ein **"Hello, World!"**-Programm reichlich 50 KByte groß ist, weil der Linker gnadenlos riesige Bibliotheken komplett einbindet.

Das ist es, was man allgemein von einem Linker erwartet, zumal die heute gebräuchlichen Linker immer noch auf Konzepten der 50er Jahre basieren, wie Stroustrup einmal sagte. Auch der Standard-Linker des Amiga, "*ALink*", die verbesserte Version "*BLink*" oder die diversen Nachbauten dieser Linker tun im großen und ganzen nichts anderes. Auch der integrierte Linker von MaxonC++, der zu ALink/BLink kompatibel ist, beherrscht diese beiden Aufgaben, wenn auch zum Teil etwas eigenwillig. Nun reden wir hier aber von C++ und nicht von solchen Antiquitäten wie Fortran oder C, und die neue Zeit fordert nicht nur neue Compiler, sondern stellt auch neue Anforderungen an die damit verbundenen Linker.

Eine der zahlreichen, auf den ersten Blick eher unscheinbaren Neuerungen, die C++ gegenüber C so auf dem Kasten hat, ist die Initialisierung statischer Daten. In C dürfen globale Variablen ausschließlich mit konstanten Werten initialisiert werden, weshalb eine Deklaration wie

```
int i = 42;
```

weder Compiler noch Linker vor unlösbare Probleme stellt: Die Variable `"i"` wird im Datenhunk des Programms angelegt, und die entsprechende Speicherstelle bekommt von Anfang an den Wert `"42"`. Schon in dem Augenblick, wenn das Programm in den Speicher geladen wird, steht also an der Stelle, an der sich später einmal die Variable `"i"` befinden wird, das entsprechende Langwort, und das Schönste daran ist, daß das den Linker überhaupt nicht zu interessieren braucht, denn er bekommt auf diese Weise gar nicht mit, daß es eine solche Initialisierung gibt.

In C++ ist die ganze Angelegenheit schon etwas fortschrittlicher, denn dem Programmierer werden in seinen Initialisierungsdrang keinerlei Zügel mehr angelegt. Das folgende ist zum Beispiel ein korrektes C++-Programm:

```
#include <stdio.h>

int murx = printf("Sachen gibt\'s!!!");

void main() { }
```

Obwohl das Hauptprogramm **"main"** scheinbar leer ist, macht dieses Programm etwas, denn noch vor der Ausführung von **"main"** wird die globale Variable **"murx"** mit dem Ergebnis eines **"printf"**-Aufrufs initailisiert (das Ergebnis von **"printf"** ist die Anzahl der ausgegebenen Zeichen oder ein Fehlercode). Also passiert hier schon eine Ausgabe, bevor überhaut das scheinbare Hauptprogramm gestarten wird.

In dieser Form ist das natürlich ein eher krankhafter Fall von Programmverschleierung, aber durch Konstruktoren kommt es in C++ oft zu solchen absolut unkonstanten Initialisierungen. Die ganz besondere Pointe ist, daß jedes Modul solche Initialisierungen besitzen kann. Lange Rede, schwacher Sinn: Der Linker muß in C++ nicht bloß linken, sondern auch dafür sorgen, daß Module korrekt initialisiert werden - ja, nicht nur das: falls globale Variablen einen Destruktor haben, muß er sogar dafür sorgen, daß nach Programmende derartige Destruktoren aufgerufen werden.

Sie werden lachen, aber der MaxonLINKER kann das! Wie das genau vor sich geht, wird an anderer Stelle erläutert. Kommen wir statt dessem noch einmal auf die beiden anfangs erwähnten Aufgaben des Linkers zu sprechen.Wie Sie MaxonC++ dazu bringen, Module zusammenzubinden, dürfte aus den vorhergehenden Kapiteln hoffentlich klargeworden sein (Nämlich in der integrierten Umgebung mit einem Makefile und beim "*mcppc*" wahlweise mit einem Makefile oder entsprechenden Kommandozeilen-Argumenten). Deshalb widmen die beiden folgenden Abschnitte sich der anderen Funktion, dem Einbinden von Bibliotheksfunktionen:

4.2 Standard-Funktionen und das "Logfile"

Der Linker von MaxonC++ beherrscht das Einbinden von Bibliotheksroutinen erstaunlicherweise gleich auf zwei verschiedene Arten. In diesem Abschnitt soll von der etwas unkonventionelleren Methode die Rede sein, nämlich dem Linken mit Objekt-Dateien über eine Datei namens "*Logfile*".

Im großen und ganzen ist eine Bibliothek nichts anderes als eine Sammlung von Modulen mit nützlichen Funktionen. Der Linker geht nun hin und nimmt sich selbsttätig aus dieser Sammlung alle Module, die gerade benötigt werden, und benötigt werden eben jene Objekt-Dateien, in denen Symbole (also z. B. Funktionsnamen) deklariert werden, die im aktuellen Programm benutzt werden, bisher aber nicht definiert wurden.

Es ist deshalb ein naheliegender Gedanke, dem Linker in Form einer Datei mitzuteilen, welche Objektdatei er einlinken soll, wenn ein bestimmtes Symbol sonst nicht bekannt ist. Genau dieses genial einfache, aber zugegebenermassen nicht sehr kompatible Prinzip wurde bei MaxonC++ imple-

mentiert: Im "*lib*"-Verzeichnis, in dem sämtliche Linker-Libraries stehen, gibt es eine Datei namens "*logfile*", welche Informationen darüber enthält, was wo steht. Der Dateiname wurde zugegebenermaßen etwas eigenartig gewählt, denn normalerweise versteht man unter einem "*log*" so etwas wie ein Protokoll - jeder TeX-User kennt die "*.log*"-Files. Hier ist das "*log*" ganz einfach eine legasthenische Verkürzung von "Katalog". Wenn ich ehrlich sein soll, habe ich das "*logfile*" so genannt, weil das einfach gut klingt und mir gerade nichts besseres einfiel. Wesentlich interessanter dürfte für Sie sein, was in dieser Datei enthalten ist.

Das "*logfile*" ist eine ganz gewöhnliche Textdatei, die man leicht mit jedem Editor bearbeiten kann, und besteht aus einer Folge von Einträgen wie

```
Dateiname1
   Symbol1
   Symbol2
   Symbol3
Dateiname2
   Symbol4
usw.
```

Eine Zeichenfolge, die in der ersten Spalte beginnt, stellt also den Namen einer Objektdatei dar, die sich nach Möglichkeit ebenfalls im "*libs*"-Verzeichnis befinden sollte. Falls jedoch im Dateinamen ein `":"` oder `"/"` vorkommen sollte, kann die Objektdatei liegen, wo sie will, Hauptsache der Pfad wurde auch richtig angegeben.

Jedem Dateinamen kann eine nahezu beliebige Folge von Symbolnamen folgen, die sich dadurch auszuzeichnen haben, daß sie in ihrer Zeile um mindestens ein Leerzeichen eingerückt sind. In diesem Zusammenhang sei davor gewarnt, hier <TAB>´s zu benutzen, denn die mag der Linker überhaupt nicht.

Zur Steigerung der Anschauung hier ein Auszug aus dem originalen Logfile:

```
startup.o
   __StartUp
dosbase.o
   exit_i
   _exit
   abort_standard
   _DosBase
   _SysBase
   _Filehandle_Std_Input
   _Filehandle_Std_Output
   _cin
   _cout
   _cerr
   _std__in
   _std__out
   _std__err
   _errno
   __Writefile
   __Writefile__NoBuf
```

```
wbstartup.o
   _wbmain
stdinput.o
   _getc
   getc__P06stream
   _fgetc
   fgetc__P06stream
   _getchar
   getchar_
   __Readfile
   __Readfile__NoBuf
arg.o
   main_
```

Es gibt hier einige (übrigens in Assembler geschriebene) Objektdateien, in denen bestimmte Linker-Symbole deklariert werden, zumindens behauptet das Logfile das. Wenn also in einem Programm ein Symbol namens `"_getc"` referiert wird, weiß der Linker, daß er die Datei "*stdinput.o*" aus dem "*libs*"-Verzeichnis einbinden soll.

Es gibt im wesentlichen für jede Funktion eine eigene Objektdatei. Oft wurden aber aus rein praktischen Erwägungen Funktionen mit gemeinsamen Unterroutinen (z. B. `"getc"` und `"fgetc"`) oder sinngemäß zusammenhängende Funktionen (kein `"fopen"` ohne `"fclose"`) in einer Objektdatei zusammengefaßt. Effektiv wird hier also nur das eingebunden, was auch tatsächlich benutzt wird.

Damit dürfte auch klar sein, wie Sie selbst die Standard-Bibliotheken erweitern können: Kopieren Sie einfach die Objektdatei in das "*libs*"-Directory, und fügen Sie dem Logfile einen entsprechenden Eintrag hinzu. Wenn Sie sich nicht sicher sind, wie die Linkersymbole zu ihren Routinen heißen - was in C++ ja nicht so ganz trivial ist, da ja die Parameterlisten in den Symbolnamen hineincodiert werden, gibt es zwei Möglichkeiten, diese herauszufinden: Entweder lassen Sie sich ein Assembler-Listing erzeugen und suchen sich dort die `"XDEF"`-Zeilen heraus, oder Sie versuchen zuerst einmal "einfach so", die selbstgeschriebenen Routinen einzubinden, und merken sich dann, was der Linker nicht finden kann.

4.3 Das Hauptprogramm und seine Argumente

Wenn Sie sich das Logfile durchlesen, erweckt vielleicht der Eintrag `"main_"` für die Datei "*arg.o*" Ihr berechtigtes Interesse, auch wenn dieser leider eben nicht besagt, daß man das Hauptprogramm überhaupt nicht mehr selbst schreiben muß. `"main_"` ist in C++-Linkage der Name des parameterlosen Hauptprogramms. Nun darf man `"main"` natürlich auch mit den beiden kanonischen Argumenten deklarieren, also so:

```
int main(int, char**)
```

Wird dies im C++-Modus übersetzt, lautet das zugehörige Linkersymbol `"main_iPPc"`, und im C-Modus wird der Programmname für den Linker immer in `"_main"` übersetzt.

Dieses Dilemma löst MaxonC++ elegant dadurch, daß der Startup-Code zunächst in eine Funktion `"main_"` einspringt. Wenn der Linker dieses Symbol nicht findet, liegt das entweder daran, daß der Programmierer das Hauptprogramm schlicht vergessen hat, oder es ist eben darauf zurückzuführen, daß `"main"` mit Parametern oder im C-Modus deklariert wurde. Dem Linker ist dies zunächst einerlei, und da das Symbol `"main_"` im Startup-Code referenziert, aber nirgendwo definiert wird, sucht er es wie jedes andere Symbol in den Bibliotheken. Dort wird er auch prompt fündig und bindet die Objektdatei "*arg.o*" ein. Hinter dem Symbol `"main_"` verbirgt sich in diesem Bibliotheksmodul eine kleine Routine, die den Argumentvektor `"argv"` erzeugt und initialisiert, `"argc"` und den Zeiger auf `"argv"` auf den Stack ablegt und dann in eine Routine `"main_iPPc"` einspringt.

Wurde das Hauptprogramm im C++-Modus mit den üblichen Parametern deklariert, trägt es eben jenes Symbol, und die Sache ist für den Linker gegessen. Im C-Modus muß er noch einen weiteren Schritt machen: er bindet zusätzlich das Modul `"another_main.o"` ein, in dem ein Symbol `"main_iPPc"` deklariert wird. In dieser Mini-Datei steht nichts anderes als ein Sprung an eine Adresse namens `"_main"`, und auf diese Art und Weise wird auch in C-Programme korrekt eingesprungen.

Das Angenehme an dieser Rumlinkerei ist, daß auf diese Weise der Code für die Berechnung von `"argv"` in C++ nur dann eingebunden wird, wenn das Programm die Argumente auch auswertet, das spart einige Bytes an Codegröße. Im C-Modus wird dieser Code leider immer eingebunden, und hinzu kommen noch 8 Bytes sowie ein Relokationstabelleneintrag für den zusätzlichen Sprung.

Ach ja, noch was: Es gibt auch eine Bibliotheksfunktion `"_wbmain"`. Wenn Sie also nicht selbst eine Startup-Routine für die Workbench schreiben, nimmt der Linker die aus der Bibliothek. Diese Funktion macht aber nichts anderes als umgehend auszusteigen. MaxonC++ geht davon aus, daß Programme, die nicht eigens dafür geschrieben sind, von der Workbench gestartet zu werden, nicht künstlich Workbench-fähig gemacht werden sollten (etwa durch zwangsweises Öffnen eines Text-Fensters oder ähnlichen Schnickschnack). Daß ein MaxonC++-Programm also nicht ohne weiteres von der Workbench startbar ist, ist kein Bug, sondern ein Feature.

Näheres zu der Frage, wie man ein Programm Workbench-tauglich macht, steht übrigens im Kapitel 6.2 des Tutorials.

4.4 Linker-Bibliotheken

Ich glaube, ich erwähnte schon, daß die Sache mit dem Logfile eine Spezialität von MaxonC++ ist. Der große Vorteil ist dabei, neben der einfachen und flexiblen Erweiterbarkeit, die recht hohe Geschwindigkeit vor allem bei langsameren Speichermedien: Der Linker hat detaillierte Informationen darüber, was er einbinden soll, und muß dann nur noch die benötigten Dateien laden.

Gebräuchlicher sind hingegen Linker-Libraries, bei denen die gesamte Bibliothek in einer Datei zusammengefaßt wird. Der Linker liest sich diese Datei aufmerksam durch und pickt sich die benötigten Teile heraus. Interessant ist dieses Verfahren vor allem deshalb, weil die offiziell von Commodore vertriebene "*amiga.lib*" in einem solchen standardisierten Format vorliegt - natürlich alles "ALink/BLink"-kompatibel.

Aus diesem Grund unterstützt auch der MaxonC++ Linker auch dieses Bibliotheksformat. Man kann eine beliebige Anzahl solcher Bibliotheken angeben, aus denen er dann alles nimmt, was er sonst nicht kriegen kann.

Rein technisch erfolgt die Deklaration der zu untersuchenden Bibliotheken in der Kommandozeilenversion mit der Option `"-l"` und in der integrierten Umgebung mit einem entsprechenden Eintrag im entsprechenden Symbol des Linker-Dialogfensters. Auch in der Makedatei kann man die gewünschten Bibliotheken in der zweiten Zeile (der Linker-Anweisung, die mit "*mcppc*" anfängt) angeben.

4.5 Initialisierungen und das große Aufräumen

In 4.1 wurde schon mit einem gewissen Stolz darauf hingewiesen, daß der MaxonC++ Linker als extraordinäres Feature auch Module initialisieren kann. Da fragt sich der geneigte Leser natürlich, woher dieses Wunderprogramm weiß, was wie initialisiert werden soll.

Die Lösung dieses Problems ist ganz einfach: Wenn der Linker ein Symbol findet, das mit `"_INIT_x_"` anfängt - wobei `"x"` eine Ziffer zwischen `0` und `9` ist - weiß er, daß diese Funktion im Startup-Code aufgerufen werden soll. Umgekehrt werden Funktionen mit dem Anfang `"_EXIT_x_"` am Programmende aufgerufen. Die Ziffer `"x"` gibt in beiden Fällen die Priorität der Initialisierung bzw. des Cleanups an: Initialisierungen mit der Nummer `0` werden als erste aufgerufen, die mit `9` als letzte. Bei den

`"_EXIT_"`-Funktionen ist die Reihenfolge umgekehrt. Wenn der Compiler ein Modul erzeugt, das initialisiert werden muß, erzeugt er eine Initialisierungsfunktion mit der Priorität 8, und für eventuelle Destruktoraufrufe wird entsprechend eine Exit-Funktion gleichen Vorrangs generiert.

Der genaue Name der Init- und Exit-Symbole ist belanglos, sie sollten nur im Programm möglichst eindeutig sein. Deshalb kodiert der Compiler den Namen der Objektdatei in die Funktionsnamen, während in den fertigen Libraries (Sie erinnern sich: alle in Assembler geschrieben) möglichst seltsame Namen gewählt wurden, die bestimmt nirgendwo sonst vorkommen, z. B. `"_INIT_0_files_o_Yeah_Initialise_me_Babe"` im Modul "*files.o*".

Der Linker erzeugt selbstständig einen kleinen Startup-Code, der erst sämtliche Initialisierungen in der Reihenfolge ihrer Priorität aufruft und dann in den eigentlichen Startup-Code, der für gewöhnlich aus dem Bibliothksmodul "*startup.o*" eingebunden wird, einspringt. Außerdem generiert der Linker stets eine Funktion, in der sämtliche Exit-Funktionen aufgerufen werden. Die Standard-Funktion `"exit"` springt dieses Label an.

4.6 Mit und ohne Startup-Code

Manchmal will man den oben beschriebenen Startup-Code nicht haben, zum Beispiel, weil man eine Shared Library programmieren will oder weil einem der Startup-Code einfach nicht paßt. Deshalb gibt es einige Linkeroptionen, die eben diesen unterdrücken.

Zuerst wäre da die schlichte *CPPC*-Option `"-gs"` oder die Einstellung "***Ohne Startup-Code***" im "**Linker**"-Dialogfenster von "*MCPP*". Bei dieser Option werden die Module des Programms in der angegebenen Reihenfolge gelinkt, die benötigten Bibliotheksroutinen werden angehängt und als allerletzter Hunk wird eine Sparversion des Startup-Codes erzeugt. Dieser kleine Hunk enthält dann zwei parameterlose Funktionen mit den Labels `"_InitModules"` und `"_CleanupModules"`. Diese sollte man auf jeden Fall am Programmanfang bzw. -ende aufrufen. Von C++ aus muß man sie natürlich in C-Linkage importieren, z. B. so:

```
extern "C"
{ void InitModules();
  void ExitModules();
}
```

Diese Definitionen stehen aber auch im Include-File "*<linkerfunc.h>*".

Falls man, das kanonische Beispiel für Linken ohne Startup-Code, eine Shared Library implementiert, sollte `"InitModules()"` die allererste Anweisung in der Open-Routine der Library und `"CleanupModules()"` entsprechend die letzte Anweisung in der Schließroutine sein.

Für fanatische Bytezähler, aber auch für Leute, die "*mcppc*" als BLink-Ersatz verwenden möchten, hat die CLI-Version noch zwei schlauere Optionen. Zum einen gibt es da noch `"-gsi"`. Hier wird der kleine Zusatz-Hunk mit den Funktionen `"InitModules"` und `"ExitModules"` nur dann erzeugt, wenn eines der beiden Labels auch tatsächlich irgendwo im Programm referiert wird. Dabei sind rätselhafte Programmfehler natürlich schon fast vorprogrammiert, denn man sollte die beiden Funktionen eigentlich vorsichtshalber immer aufrufen, da man ja nie wissen kann, ob irgendwelche Library-Module eine Initialisierung benötigen.

Deshalb sollten Sie `"-gsi"` ganz schnell wieder vergessen und stattdessen die Option `"-gsw"` verwenden. Diese funktioniert ganz genau wie `"-gsi"`, gibt aber eine Warnung aus, wenn eine Initialisierung bzw. ein Cleanup nötig ist, aber nicht durchgeführt wird. Im Klartext: Wenn irgendein Modul eine `"_INIT_x_"`-Funktion besitzt und das Label `"_InitModules"` nirgendwo im Programm referiert wird, gibt der Linker so etwas wie **„_Symbol '_InitModules' never used“** aus - für den Cleanup analog. Die Option `"-gsw"` ist recht praktisch, wenn man sich recht sicher ist, daß nirgendwo im ganzen Programm einschließlich der Standard-Bibliotheken eine dynamische Initialisierung nötig ist, die entsprechenden Funktionsaufrufe deshalb wegläßt und bei Bedarf über seinen Irrtum aufgeklärt werden möchte.

Für echte Freaks ist es natürlich unter ihrer Würde, auf fertige Routinen zurückzugreifen, und so ist ihnen auch der normale Startup-Code suspekt. Es ist nicht weiter schwierig, sich einen völlig eigenen Startup-Code zu schreiben. Linkt man ein Programm ohne Startup-Code und startet es dann,

4.7 Der Startup-Code

Für alle, die genauer wissen möchten, was beim Programmstart passiert, folgt hier eine kommentierte Fassung des Quelltexts für das Bibliotheksmodul "*startup.o*". Wenn man einmal die unvermeidlichen Symbol- und Konstantendefinitionen wegläßt, bleiben ganze drei Funktionen übrig. Hier ist die erste:

```
_INIT_0_Get_StartuP_MessAge:
        move.l sysbase.w,a6
        move.l $114(a6),a5
        tst.l  $AC(a5)
        bne.b  ret
        move.l #__Writeflag,d0
        bne.b  ret
        lea    $5C(a5),a0
        move.l a0,-(a7)
        jsr    WaitPort(a6)
        move.l (a7)+,a0
        jsr    GetMsg(a6)
        move.l d0,StartupMessage
ret     rts
```

Diese Init-Routine mit Priorität 0 ist garantiert das erste Stück Code, das im Programm ausgeführt wird - es sei denn, ein heimtückischer Assembler-Programmierer fügt eine Init-Routine gleicher Priorität hinzu. Wie der Amiga-Kenner leicht sieht, wird hier als erstes geprüft, ob das Programm von der Workbench oder aus der Shell gestartet wurde. Trifft ersteres zu, wird die Startup-Message abgeholt und in einer Variablen namens `"StartupMessage"` abgelegt.

Das seltsame Symbol namens `"__Writeflag"` ist übrigens ein übler Hack, durch den die Umgebung *MCPP* Zugriff auf die Standard-Ein- und -Ausgabe des Programms erhält. *MCPP* trägt dort die Adresse einer trickreichen Routine ein, während es bei einem "richtig" gelinkten Programm immer Null ist. Daß dieser Wert in der obigen Routine abgefragt wird, war bei antiquierten Versionen von *MCPP* noch nötig. In absehbarer Zeit wird dieser Test wohl aus dem Startup-Code 'rausfliegen.

Der kurze Code, den der Linker im normalen Modus an den Anfang des Programms setzt, ruft also zuerst alle Init-Routinen in der richtigen Reihenfolge auf und springt dann in die Routine `"__Startup"` ein. Wenn man sie nicht überdefiniert, nimmt er dafür die folgende:

```
__StartUp:
        move.l StartupMessage,d0
        beq.b  .cli
        move.l d0,-(a7)
        jsr    _wbmain
        addq.l #4,a7
        moveq  #0,d0
        rts
.cli    jmp    main_
```

wird in die erste Routine in der ersten Übersetzungseinheit eingesprungen. Es kommt dann also beim Linken wirklich auf die Reihenfolge der Module an. Durch geschickt definierte Registerparameter kann eine solche Funktion sogar die Shell-Argumente auswerten. Zuvor sollte sie aber die Module initialisieren und im Zweifelsfall mit `"GetBaseReg()"` den Zeiger auf den Small-Data-Hunk initialisieren.

Das Beispielprogramm "*nostartup.c*" demonstriert dies:

```
// nostartup.c - Demo für Linken ohne Startup-Code//
// Compileraufruf: cppc nostartup.c -gs
// bzw. "Ohne Startup" im M+Plus-Linker-Requester einstellen

#include <linkerfunc.h>
#include <stream.h>

// Die erste Funktion im ersten Hunk wird angesprungen:
void f(register __d0 int len, register __a0 char *par)
// In Registern d0 und a0 werden die Shell-Parameter übergeben
// Diese Register-Belegungen bremsen die Funktion übrigens gut aus,
// deshalb sollte man in dieser Funktion nicht mehr als nötig tun.
{
  // Wichtig, wenn Small Data-Hunks existieren, und sonst auch
  // nicht verkehrt:

  GetBaseReg();    // Initialisierungen, z. B. Dos-Lib öffnen:
  InitModules();
  par[len-1]=0;       // Normalerweise kein
                        // Nullbyte am Ende
  // Jetzt geht alles wie immer:
  cout << "Es geht auch ohne Startup-Code!" <<
  "\nParameter-Länge: " << len << "\nParameter: " << par
  << "\n";
  // Programmende: Aufräumen, z. B. DosLibrary schlie_en:
  CleanupModules();

  // "exit()" darf ohne Startup-Code nicht benutzt werden!!!
  // Es ist nur so ein sinnvoller Return-Code möglich:
  len = 42;
}
```

Es handelt sich also um eine simple Fallunterscheidung, die entweder in `"main_"` oder `"_wbmain"` einspringt. Diese beiden Funktionen wurden schon in Abschnitt 4.3 besprochen.

So, jetzt bleibt nur noch eine kurze Funktion:

```
_EXIT_0__Reply_StartuP_MessAge:
        move.l  StartupMessage,d2
        beq.b   ret
        move.l  sysbase.w,a6
        jsr     Forbid(a6)
        move.l  d2,a1
        jmp     ReplyMsg(a6)
```

Durch die Priorität 0 ist dies die letzte Funktion, die am Programmende aufgerufen wird. Wurde das Programm von der Workbench gestartet, wird hier die Startup-Message beantwortet, und das wars dann schon.

Assembler-Programmierer können aus diesen Listings auch noch die Erkenntnis gewinnen, daß Init- und Exit-Funktionen offensichtlich keine Rücksicht auf das Trashing von Registerwerten nehmen müssen.

4.8 Hunks von Aufteilen, Zusammenfassen und Wegschmeißen

An dieser Stelle soll noch einmal an drei besondere Features des MaxonC++ Linkers erinnert werden - nur damit Sie sich nicht wundern, wenn Sie die Hunks Ihrer Objektdateien im ausführbaren Programm nicht mehr so wiederfinden, wie Sie es vielleicht erwartet haben.

Zum einen ist da die Eigenschaft, daß unreferenzierte Hunks grundsätzlich nicht in die Exe-Datei aufgenommen werden. Dies erwies sich in der Version 3.0 als nahezu unverzichtbar, da bei Templates "auf Verdacht" jede Menge Code angelegt werden muß, von dem am Ende vieles nicht gebraucht wird. Obendrein werden diese Funktionen oft in mehreren Übersetzungseinheiten zugleich angelegt, was ja nun wirklich nicht wein muß. Deshalb wurde eine Konvention eingeführt: Alles, was in einer Objektdatei ab dem zweiten Codehunk steht, wird wie ein Library-Modul behandelt, d. h. Überschneidungen mit identischen Symbolen aus anderen Objektdateien werden ignoriert.

Dieses großzügige Hunk-Wegwerfen war Voraussetzung für den Library-Modus, der strenggenommen ein Compiler-Feature ist. Wie schon in Abschnitt 2.5.3.2 beschrieben, wird bei der Codeerzeugung für jede Funktion und jede globale Variable ein eigener "Hunk" erzeugt. Der Linker schmeißt kann nicht benutzte Funktionen platzsparenderweise heraus.

Und zu guter letzt sei noch einmal an das optionale Zusammenfassen von Hunks erinnert (2.5.3.5). Dadurch sehen die ausführbaren Programmdateien wesentlich aufgeräumter aus, und ein paar Bytes spart es auch.

5. Die CLI-Version "MCPPC"

5.1 Was ist neu?

Wenn Sie bereits die 1.x-Version von Maxon C++ kennen, wird Sie als erstes sicher interessieren, was das vorliegende Update Neues bringt. Sehen wir einmal davon ab, daß alles besser und schöner geworden ist, so kommen wir auf folgende Feature-Liste:

- Bei Make-Dateien ist natürlich das in Abschnitt 3.3 vorgestellte, erweiterte Format erlaubt.
- In den Make-Dateien können die Abhängigkeiten vollautomatisch gesetzt werden.
  Optionen `"-mx"` und `"-mX"`
- Es gibt drei verschiedene Formate bzw. Modi für Fehlerausgaben: mit `"-ei"` schaltet man in den bekannten interaktiven Modus (mit Nachfrage nach jedem Error), `"-e<N>"` ist wie gehabt die Option für die Kurzausgabe und mit `"-el<N>"` erreicht man das neue Langformat. Die optionale Angabe `"<N>"` steht dabei für die maximale Anzahl von Meldungen.
- Der Quelltext bzw. die Positionsangabe können zwecks besserer Übersichtlichkeit in beliebiger Farbe ausgegeben werden.
  Option `"-ec<N>"`
- Die eigentliche Fehlermeldung kann auf Wunsch fett („bold") hervorgehoben werden.
  Optionen `"-eb"` und `"-eB"`
- Fehlermeldungen können in einer Datei abgelegt werden.
  Option `"-ef"`
- Warnungen können wie Fehler behandelt und leichter eingeschaltet werden.
  Optionen `"-ew"` bzw. `"-w+"`
- Das Exception Handling (also eines der wesentlichen neuen Compiler-Features) kann ein- oder ausgeschaltet werden.
  Optionen `"-px"` und `"-pX"`
- Beim Erzeugen von Template-Funktionen sind zwei verschiedene Strategien möglich.
  Optionen `"-t"` und `"-T"`
- Code für die 32Bit-Prozessoren (ab 68020) und Coprozessoren kann erzeugt werden.
  Optionen `"-g00"` und `"-gs0"` bzw. `"-g80"` und `"-g81"`
- Echt vorcompilierte Header-Dateien können gelesen und abgespeichert werden.
  Optionen `"-h0"`, `"-h <Name>"` und `"-H <name>"`
- Im Library-Modus wird für jede Funktion und jede Variable ein separater Hunk erzeugt.
  Optionen `"-gl"` und `"-gL"`

☞ Zu jedem Switch gibt es jetzt eine Option, mit der man explizit die Default-Einstellung einschalten kann.

☞ Für einige wichtige Optionen wurden Abkürzungen eingeführt:

`-O`	`-gO`	Optimieren
`-L`	`-gL`	Library-Modus
`-X`	`-pX`	Ausnahmebehandlung zulassen

5.2 Das Konzept

Es soll angeblich tatsächlich Leute geben, denen dir integrierte Entwicklungsumgebung von MaxonC++ nicht gefällt. Deshalb gibt es eine CLI-Version von MaxonC++, nämlich das Programm "MCPPC".

Eigentlich ist MCPPC ebenfalls ein integriertes System, denn es enthält Compiler, Assembler und Linker. Der Unterschied zum "MCPP" liegt also ausschließlich in der Benutzerschnittstelle: MCPPC funktioniert wie ein Shell-Befehl und wird über Kommandozeilen-Argumente gesteuert.

Das System wurde ähnlich wie die C-Compiler unter Unix konzipiert, d. h. compilieren, assemblieren und linken erfolgt mit einer einzigen Anweisung (oder optional auch getrennt). Dateinamen, die auf ".o", ".obj" oder ".l" enden, werden als Objektfiles interpretiert und eingelinkt, alles andere betrachtet der Compiler als Namen von Quelltexten.

Ohne besondere Optionen übersetzt der Compiler alle angegebenen Quelltexte (ohne die entsprechenden ".o"-Files zu schreiben), linkt die angegebenen Objektdateien ein und erzeugt eine ausführbare Datei, deren Name sich vom jeweils ersten Dateinamen ableitet:

```
mcppc test.o haupt.c text.c unter.c nochwas.o
```

compiliert die drei Quelltexte "*haupt.c*", "*text.c*" sowie "*unter.c*", erzeugt aber keine Objektdateien, sondern hält das Ergebnis dieser Übersetzungen resident im Speicher. Anschließend lädt er die beiden Objektdateien "*test.o*" und "*nochwas.o*" dazu und erzeugt daraus eine ausführbare Datei namens "*test*".

Bei kleinen Programmen, die aus nur einem Modul bestehen, ist der Compiler also ausgesprochen einfach zu bedienen: Die Anweisung

```
mcppc name.c
```

übersetzt das Programm, linkt es und erzeugt eine ausführbare Datei namens „*name*". Einfacher geht es wohl nicht.

Tritt bei der Übersetzung ein Fehler auf, wird ein geeigneter kurzer Auszug aus dem Quelltext ausgegeben, zusammen mit einer Ausgabe der Art

```
Datei "Dateiname.c", Zeile 17:
Error: Identifier "ths" not defined.
Weiter mit <Return>, abbrechen mit <A>:
```

Jetzt können Sie also entweder mit der <Return>-Taste den Fehler übergehen und die Übersetzung fortsetzen, oder <A> tasten und damit den Compiler abbrechen.

MCPPC ist residentfähig, auch wenn einige Shells einen Prüfsummenfehler melden. Dies ist in jedem Fall harmlos, auch wenn *MCPPC* auf eine etwas ungewöhnliche Methode reentrant gemacht wurde: Anhand eines Semaphors sorgt eine geladene Kopie des Programms dafür, daß sie nicht zweimal gleichzeitig läuft.

5.3 Optionen

5.3.1 Allgemeines

Die oben beschriebenen Features allein sind natürlich noch etwas mager. Alle darüber hinausgehenden Funktionen werden durch Compileroptionen gesteuert, die in der Kommandozeile angegeben werden und jeweils mit einem Minuszeichen `"-"` beginnen.

Um ein wenig Ordnung in dieses umfangreiche Sammelsurium zu bringen, lassen die Optionen sich anhand ihres jeweils ersten Buchstaben in verschiedene Bereiche einteilen:

- `-a` Assembler-Sourcegenerierung
- `-b` Debugging
- `-c` Wahl zwischen Exe- und Objektdatei
- `-d` Symboldefinitionen
- `-e` Fehlermeldungen
- `-g` Codegenerierung
- `-h` vorcompilierte Header-Dateien
- `-i` Pfade für Includes
- `-j` Pfade für Libraries
- `-l` Linker-Libraries
- `-m` Make
- `-o` Ausgabedatei
- `-p` Parser
- `-w` Warnungen und Workspacegröße

Optionen sind stets global: Bei einer Zeile der Form

```
mcppc -x text1.c -y text2.c -z
```

werden "*text1.c*" und "*text2.c*" gleichermaßen mit den Optionen "*x*", "*y*" und "*z*" compiliert. Mit anderen Worten ist die Reihenfolge von Dateinamen und Optionen egal.

Die Reihenfolge der Optionen untereinander ist aber nicht egal, wenn in einer Anweisungszeile einander widersprechende Optionen angegeben auftreten. Beispielsweise schaltet man mit `"-go"` die Optimierung ein und mit `"-go"` aus. In Fällen wie

```
mcppc -go -g0 usw.etc
```

gilt die jeweils letzte Angabe.

Nicht nur für die Optimierung, sondern für fast jedes Feature gibt es zwei Optionen: eine zum Einschalten und eine zum Ausschalten. Das erscheint auf den ersten Blick überflüssig, ist doch z. B. die

Optimierung standardmäßig ohnehin nicht aktiviert und damit eine explizite Ausschaltoption eigentlich nicht erforderlich. Man kann aber alle diese Grundeinstellungen ändern, und zwar mit der Umgebungsvariablen `"CPPOPT"`, zu der wir später kommen. Wenn Sie damit `"-g0"` zum Default machen, wollen Sie vielleicht doch noch einmal mit `"-go"` zurückschalten.

Bei den meisten Umschaltmöglichkeiten wurde der Weg gewählt, daß man mit einem Großbuchstaben ein- und mit einem Kleinbuchstaben ausschaltet, so wie bei `"-go"` und `"-g0"`. In einigen Fällen, vor allem bei weniger oft benötigten Features, gab es in den 1.x-Versionen von Maxon C++ nur einen Einschalter mit lauter Kleinbuchstaben und keine explizite Rückschaltmöglichkeit. Das wichtigste Beispiel ist wohl die Option `"-b"` für die Erzeugung von Debugger-Dateien. Aus Gründen der Kompatibilität (wer gewöhnt sich schon gerne um, vor allem bei solchen ohnehin kryptischen Kürzeln) wurden diese Optionen so belassen und die neuen Ausschalter durch Anhängen einer Null definiert. `"-b0"` bedeutet also "Keine Debugger-Dateien schreiben".

In einigen Fällen wurden auch Abkürzungen eingeführt, z. B. ist `"-O"` identisch mit `"-go"`. Okay, eine Zwei-Buchstaben-Kombination durch einen einzelnen Buchstaben abzukürzen, ist vielleicht keine besonders eindrucksvolle Ersparnis. In diesem speziellen Fall hat `"-O"` aber auch die Vorteile, schöner auszusehen und zu einigen anderen Compilern kompatibel zu sein. Was will man mehr?

5.3.2 Ergebnisse und Ausgaben

5.3.2.1 Assembler-Quelltext: -a und -as

Vielleicht möchten Sie sich einmal den von Maxon C++ erzeugten Code ansehen, oder Sie gehören zu den Leuten, die compilierte Programme von Hand nachoptimieren. Mit der Option "-a" können Sie sich dann ein Assemblerlisting ausgeben lassen.

```
mcppc test.c -a
```

erzeugt beispielsweise nicht nur die ausführbare Datei "*test*", sondern auch das Assembler-Listing "*test.s*". Die Angabe `"-a"` bricht den Übersetzungsvorgang also nicht vor der Assemblierung ab, sondern erzeugt den Quelltext zusätzlich zu den übrigen Aktionen.

Ein reines Assemblerlisting ist aber meist ziemlich unübersichtlich. Deshalb gibt es die Variante `"-as"`, durch die Auszüge aus dem originalen Quelltext an den jeweils entsprechenden Stellen als Kommentar in den Assembler-Quelltext eingefügt werden.

Um diese beiden Optionen explizit auszuschalten, verwenden Sie `"-a0"`.

5.3.2.2 Objektdateien: -c

Die Option `"-c"` unterdrückt das Linken (und damit auch die Erzeugung einer ausführbaren Datei) und erzeugt statt dessen eine Objektdatei:

```
mcppc -c file1.c file2.C
```

erzeugt die beiden Objektdateien "*file1.o*" und "*file2.o*" - natürlich nur, wenn die Quelltexte auch erfolgreich übersetzt werdne konnten.

Die Option `"-c"` ist der Schlüssel zur modularen Programmierung: Wenn ein Programm aus einzelnen Übersetzungseinheiten wie "*f1.c*", "*f2.c*" usw. besteht, übersetzt man mit

```
mcppc -c fN.c
```

immer nur die Module, die verändert wurden, und linkt das Ganze anschließend mit

```
mcppc f1.o f2.o f3.o ...
```

zu einer ausführbaren Datei zusammen.

Eher selten wird man dagegen die Option `"-cx"` benutzen: Dabei werden zwar auch ".*o*"-Files erzeugt, aber das Projekt wird trotzdem gelinkt. Und auch hier gibt es einen expliziten Schalter für die Grundeinstellung (also linken und schreiben der ausführbaren Programmdatei), nämlich `"-cX"`.

5.3.2.3 Ausführbare Programmdatei: -o

Wenn der Linker erfolgreich war und eine ausführbare Datei erzeugen konnte, benennt er diese normalerweise nach dem ersten Dateinamen, der in der Anweisungszeile auftrat, indem er einfach dessen Endung wegläßt. Falls das Ihnen nicht gefällt, können Sie mit der Option `"-o"` einen anderen Dateinamen angeben, z. B.

```
mcppc quelle.cpp -o Ergebnis
```

Falls mit `"-c"` auf die Erzeugung von Objektdateien umgeschaltet wurde, legt `"-o"` den Namen der Objektdatei fest.

5.3.3 Preprozessor

5.3.3.1 Definieren von Symbolen: -d

Mit der Option `"-d"` kann man "von außen" ein Preprozessor-Symbol definieren. Dabei entspricht ein Argument

```
-d NAME
```

einer Zeile

```
#define NAME
```

am Anfang des Quelltexts. Eine typische Anwendung ist eine Anweisung wie

```
mcppc text.c -d NDEBUG
```

wenn die Includedatei "*<assert.h>*" benutzt wird.

Man kann dem Symbol aber auch einen Wert zuweisen, indem man unmittelbar hinter den Bezeichner ein `"="` setzt, z. B.

```
-d ANZAHL=4711
```

Als Wert ist hier aber nur ein Token erlaubt, d. h. bei

```
-d N=17+4
```

würde das `"+4"` schlicht ignoriert.

5.3.3.2 Includedateien: -i

Der Pfad, über den CPPC die Includedateien sucht, ist mit "*MCPP:include*" vor-eingestellt. Durch einen oder auch mehrere `"-i"`-Optionen kann dieser Pfad geändert werden: Mit der Option

```
-i MCPP:inc
```

wird ausschließlich im Verzeichnis "*CPP:inc*" gesucht, und bei

```
-i MCPP:include -i MCPP:include/amiga
```

durchsucht der Preprozessor nacheinander die beiden genannten Verzeichnisse.

Tritt in einem Dateinamen ein `"|"` auf, z. B.

```
-i ram:include|df0:include
```

werden die gelesenen Dateien automatisch vom zweiten in den ersten Pfad umkopiert und dabei in ein kompakteres und schneller übersetzbares Token-Format umgewandelt, wie es am Ende von Abschnitt 2.5.2.1 beschrieben wurde.

5.3.4 Compiler-Modus

5.3.4.1 ANSI C oder C++: -pc und -pp

Maxon C++ kann durchaus auch als ANSI C-Compiler verwendet werden. Wenn Sie dazu nicht ein `"#pragma -"` an den Anfang des Quelltexts setzen wollen, können sie auch die Compileroption `"-pc"` verwenden. Default ist natürlich der C++-Modus, dem die Option `"-pp"` entspricht. Also `"c"` wie `"C"` und `"p"` wie `"Plus Plus"` - ist doch ganz leicht zu merken.

5.3.4.2 Exception Handling: -px, -pX und -X

Wie in Kapitel 8.5 des Tutorials dargestellt, hat es manchmal Vorteile, wenn man ein "klassisches" C++-Programm, das kein Exception Handling benutzt, in einem speziellen Compilermodus überset-

zen läßt. Dann werden die mit der Ausnahmebehandlung zusammenhängenden Schlüsselwörter (`"catch"`, `"throw"` und `"try"`) nicht mehr erkannt, und im erzeugten Code wird nicht über notwendige Destruktoraufrufe Buch geführt. Das bringt in C++-Programmen eine leichte Verbesserung in Bezug auf Größe und Geschwindigkeit, und bei ANSI C-Programmen hat es sowieso keine Auswirkung.

Als Default ist das Exception Handling bei *MCPPC3* ausgeschaltet, was der Option `"-px"` entspricht. Wenn Sie die Ausnahmebehandlung benutzen möchten, müssen Sie sie erst mit `"-pX"` einschalten. Als Abkürzung ist auch `"-X"` erlaubt.

5.3.4.3 Erzeugen von Template-Funktionen: -t und -T

Wie in Abschnitt 2.5.2.4.10 ausführlich dargestellt, ist es für den Compiler nicht immer leicht zu entscheiden, welche Template-Funktionen er erzeugen soll und welche nicht. `"-T"` entspricht der Strategie "alle Templates erzeugen" bzw. dem angewählten Schalter "***Alle Templates***" im "Compiler"-Dialogfenster, bei `"-t"` wird die andere Strategie verfolgt, also nur das erzeugt, was sich fehlerfrei erzeugen läßt.

5.3.5 Codeerzeugung

5.3.5.1 Breakpoints setzen: -gb

Wenn die Option `"-gb"` benutzt wird, setzt der Compiler in alle Schleifen und sonstige schleifenverdächtige Konstrukte eine Abfrage des Break-Bits, das z. B. bei Betätigung der Tastenkombination <Ctrl>+<C> gesetzt wird. Dadurch ist es möglich, so erzeugte Programme jederzeit bequem abzubrechen, allerdings kostet diese Option auch einiges an Laufzeit.

Der zugehörige Ausschalter ist `"-gb0"`.

5.3.5.2 Datenmodell: -gd und -gD

Die Option `"-gd"` schaltet auf "Small Data", `"-gD"` auf "Large Data" (siehe 2.5.3.1). "Large" ist Default.

5.3.5.3 Zusammenfassen von Hunks: -gh

Mit der Option `"-gh"` legen Sie die Höchstgrenze in KByte fest, bis zu welcher der Linker gleichartige Hunks verbindet (siehe auch 2.5.3.5), zum Beispiel: `"-gh20"` für 20 kByte. Default ist 50.

5.3.5.4 Ganzzahlige Arithmetik: -gm

Wie bereits unter 2.5.2.4.3 beschrieben, hat der 68000-Prozessor gewisse Schwierigkeiten mit der Multiplikation und Division von 32-Bit-Zahlen. Deshalb benutzt MaxonC++ dafür normalerweise korrekt rechnende Bibliotheksfunktionen, die aber recht langsam sind.

Wenn Sie das Flag `"-gm"` setzen, benutzt *MCPPC* trotzdem die normalem Rechenoperationen des Prozessors. Für den korrekten Default steht der Schalter `"-gM"`. Falls Sie aber Code für "68020-oder-höher"-Prozessoren erzeugen, ist diese Einstellung ohnehin obsolet.

5.3.5.5 Optimierungen: -go und -gO

Die zusätzlichen Optimierungen (2.5.2.4.8) werden mit der Option `"-go"` aus- und mit `"-gO"` eingeschaltet. Defaultmäßig sind sie aus. `"-O"` kann als Abkürzung für `"-go"` benutzt werden.

5.3.5.6 Code für besondere Prozessoren: -g00, -g20, -g80 und -g81

Auch wenn sich immer weniger Amiga-User mit einem schlichten 68000er-Prozessor begnügen, erzeugt MaxonC++ normalerweise Code für diesen Prozessor, der dann natürlich auch zu allen "besseren" Prozessoren kompatibel ist. Allerdings haben die Motorolas von 68020 aufwärts einige neue Befehle, mit denen Programme teilweise etwas beschleunigt werden, z. B. bei der Multiplikation und beim Arrayzugriff. Die folgenden Optionen schalten die Codeerzeugung auf einen bestimmten Prozessor um:

`-g00`	68000	(Default)
`-g10`	68010	(identisch mit -g00)
`-g20`	68020	
`-g30`	68030	
`-g40`	68040	

In der vorliegenden Version sind die Optionen `"-g20"`, `"-g30"` und `"-g40"` noch identisch, d. h. es kann nur Code für "ab 68020" und kein spezieller für 68040 erzeugt werden. Auch wenn das in späteren Compilerversionen sicher noch verbessert wird, sollten Sie sich von der CPU-Wahl keine Wunder erwarten: Wenn man sich nicht gerade ein Benchmark maßschneidert, bringen die Befehlssatzerweiterungen der 680x0-Familie nicht übermäßig viel (siehe auch Abschnitt 2.5.2.4.9).

Dagegen bringt die Benutzung der Floating Point Unit (FPU) bei Programmen, die Fließkomma-Operationen benutzen (also die Datentypen "float" und "double"), durchaus einen meßbaren Gewinn. Mit den folgenden Optionen erzeugt MCPPC Code mit oder ohne FPU:

`-g80`	ohne FPU (Default)
`-g81`	68881
`-g82`	68882 (z. Zt. identisch mit -g81)

In der Grundeinstellung, d. h. ganz ohne FPU, benutzt MaxonC++ die Fließkomma-Bibliotheken des AmigaOS, welche bei Vorhandensein einer FPU diese automatisch benutzen. Deshalb bringt auch die FPU-Benutzung letztlich nicht so viel, wie der unbedarfte Benutzer vielleicht erwarten würde.

5.3.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw

Wie in Abschnitt 4.6 beschrieben, läßt `"-gs"` beim Linken den Startup-Code weg, generiert dafür aber zwei explizit aufrufbare Funktionen `"_InitModules"` und `"_CleanupModules"`. Bei `"-gsi"` werden diese nur erzeugt, wenn sie irgendwo im Projekt referiert werden, und `"-gsw"` gibt zusätzlich eine Warnung aus, wenn eine der beiden Funktionen vorhanden ist, aber trotzdem nirgendwo aufgerufen wird.

5.3.5.7 Code für Link-Libraries: -gl, -gL und -L

Wie in 2.5.3.2 beschrieben, kann MaxonC++ auf Wunsch jede Funktion und jede globale Variable in separaten Hunks ablegen. Da der Linker unreferenzierte Hunks nicht in die ausführbare Programmdatei aufnimmt, eignet sich dieser Modus besonders für einlinkbare Bibliotheken. Mit `"-gL"` oder kurz `"-L"` wird diese Option eingeschaltet. `"-gl"` schaltet wieder in den normalen Modus zurück.

5.3.6 Debugger

5.3.6.1 Maxon Debugger: -b, -bi und -b0

Wenn Sie den Maxon Debugger besitzen, können Sie mit der Option `"-b"` die notwendigen Debugger-Dateien ("*.mdbi*" und "*.mdbg*") erzeugen lassen. Die Option `"-bi"` schreibt zusätzlich ein Icon für die "*.mdbg*"-Datei. `"-b0"` schaltet wieder in den Urzustand.

5.3.6.2 Symbolischer Debugger: -bs und -bs0

Die Option `"-bs"` erzeugt Symboltabellen-Hunks für symbolische Debugger (siehe 2.5.2.4.5).

5.3.7 Warnungen und Fehler

5.3.7.1 Format der Fehlerausgabe: -e, -el und -ei

MCPPC kann Fehlermeldungen auf drei verschiedene Arten ausgeben, nämlich interaktiv, in Kurz- und in Langform. Voreingestellt ist die interaktive Fehlerausgabe, für die auch die Option `"-ei"` steht. Dabei werden zu jeder Fehlermeldung ein kurzer Quelltextausschnitt (ca. 5 Zeilen), die Quelltextposition und eine Eingabeaufforderung ("Weiter mit <return>, Abbbrechen mit <A>:") ausgegeben. Die genaue Fehlerstelle ist im Quelltextausschnitt farbig hervorgehoben. Nach Drücken der <Return>-Taste oder alternativ der <Leertaste> oder <C> wird die Übersetzung fortgesetzt, bei Eingabe <A> wird abgebrochen. Alternativ können Sie auch mit der <ESC>-Taste oder <Ctrl>+<C> abbrechen.

Die Compileroption `"-e"` schaltet auf die Kurzausgabe. Hier werden bei jedem Fehler und jeder Warning nur zwei Zeilen mit der Quelltextposition und der eigentlichen Meldung ausgegeben. Spätestens nach zehn Fehlern wird der Übersetzungsvorgang abgebrochen. Für diese Anzahl können Sie einen beliebigen anderen Wert wählen, indem Sie die Zahl direkt an die Option anhängen, also z. B. `"-e20"` für maximal 20 Fehlermeldungen.

Das Langformat ist eine Mischung aus interaktivem Modus und Kurzform. Dabei werden Quelltext und Fehlermeldung wie bei `"-ei"` ausgegeben, ohne daß auf eine Eingabe gewartet würde. Nach maximal zehn Fehlern macht der Compiler Schluß. Für diese Form der Fehlerausgabe wählen Sie die Option `"-el"`. Auch hier kann wie bei `"-e"` auf Wunsch die maximale Fehleranzahl angehängt werden, z. B. wäre `"-el5"` eine sinnvolle Einstellung, da zehn Meldungen im Langformat in kein halbwegs normales Shell-Fenster passen würden.

5.3.7.2 Farben und Stile: -eb, -eB und -ec

Die doch etwas dröge Textausgabe können Sie optisch ein wenig aufpeppen und dabei zugleich übersichtlicher gestalten. Zum einen haben sie die Möglichkeit, die eigentlichen Fehlermeldungen mit dem Schalter `"-eB"` fett ("Bold") ausgeben zu lassen. Falls das Ihnen doch nicht so gut gefallen sollte, können Sie mit "-eb" wieder auf die Grundeinstellung zurückschalten.

Die Option `"-ecN"` bringt Farbe ins Spiel, zumindest dann, wenn Sie für `"N"` eine Ziffer einsetzen. Die Bildschirmfarbe Nummer `"N"` wird dann nämlich benutzt, um den Quelltextausschnitt von den sonstigen Ausgaben abzuheben. `"-ec1"` entspricht der Standard-Textfarbe und `"-ec0"` theoretisch der Hintergrundfarbe. Da man im letzteren Fall aber nichts lesen könnte, gilt hier die Sonderregel, daß `"-ec0"` den Quelltext in Farbe 2 ausgibt, dabei aber die Fehlerposition in einer anderen Farbkombination als `"-ec2"` hervorhebt.

5.3.7.3 Fehlerdatei: -ef

Zur Einbindung von *MCPPC* in fremde Umgebungen kann es nützlich sein, alle Fehlermeldungen in eine Datei umzuleiten. Das ermöglicht eine Angabe der Art `"-ef <name>"`. Dadurch werden alle Fehlermeldungen und Quelltextpositionen in die angegebene Datei geschrieben. Der Unterschied zu einer schlichten Ausgabeumleitung besteht darin, daß erstens `"-ef"` zusätzlich zur normalen Bildschirmausgabe arbeitet und zweitens alle Statusmeldungen nicht in der Datei erscheinen.

5.3.7.4 Optionale Warnungen: -w

Die unter 2.5.2.5 beschriebenen zusätzlichen Warnungen gibt es natürlich auch beim CPPC. Man schaltet eine Warnung ein, indem man ihren Kennbuchstaben GROß hinter das `"-w"` setzt, während ein kleiner Buchstabe eine Warnung ausschaltet.

Dies sind die Warnungen und ihre Kennbuchstaben:

- `a` (Assignment): Der Operator "=" taucht in einer Bedingung auf - vielleicht der bekannte `"=="`-Anfänger-Fehler? Sie können eine solche Warnung übrigens durch eine zusätzliche Klammerung unterdrücken, indem Sie z. B. `"if ((a=b))"` statt `"if (a=b)"` schreiben.
- `c` (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. `double -> int`
- `e` (Empty Statement): Anweisung hat keinen Code, z. B. `"42;"`
- `f` (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)
- `n` (Nested comments): Ein `"/*"` taucht in einem Kommentar auf.
- `o` (Old-Style): Über Parameterlisten im alten "K&R"-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)
- `p` (Pragma): Unbekanntes `"#pragma"`
- `r` (Return): Funktion mit Rückgabewert hat keine `"return"`-Anweisung
- `t` (Temporary): Temporäres Objekt wurde eingeführt, um ein nicht-`"const"`-Referenz zu initialisieren - was in C++ seit dem 2.0-Standard eigentlich sogar ein Error ist.

`v` (Variable): Lokale Variable wird nie benutzt, oder benutzt, aber nie initialisiert.

`m` (Macro): Ein Argument für ein Preprozessor-Makro erstreckt sich über mehr als 8 Zeilen oder 200 Token. Es liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.

Defaultmäßig sind `"-wEPRT"` aktiviert. Um z. B. Warnungen bei fehlenden Prototypen ein- und solche bei unbekannten `"#pragma"` auszuschalten, setzen Sie einfach die Option `"-wFp"` oder alternativ die beiden Optionen `"-wF -wp"`.

Um alle Warnungen einzuschalten - übrigens ein Zeichen für guten Programmierstil - ist demzufolge etwas Tipparbeit vonnöten. Um hier die Hemmschwelle möglichst niedrig zu halten, gibt es eine Abkürzung: Ein `"+"` direkt hinter dem `"-w"` schaltet alle Warnings ein. Bitte beachten Sie, daß demzufolge `"-w+v"` NICHT bedeutet, daß die Warning `"v"` zusätzlich zu denbisherigen eingeschaltet wird, sondern daß ganz im Gegenteil zuerst alle Warnings ein- und anschließend `"v"` wieder ausgeschaltet wird.

Umgekehrt können Sie auch mit `"-w-"` alle Warnungen ausschalten, wovor ich Sie aber dringendst warnen möchte. Ein sauberes Programm sollte zu überhaupt keinen Warnungen Anlaß bieten, allenfalls die `"a"`-Warnung ist eventuell als Ausdruck Ihres persönlichen Programmierstils vertretbar. Wenn Sie Warnungen löblicherweise tatsächlich so ernst nehmen, dürfte auch der folgende Abschnitt für Sie interessant sein.

5.3.7.5 Warnungen wie Fehler behandeln: -ew und -eW

Durch die Option `"-ew"` werden Warnungen wie Fehlermeldungen gewertet. `"-eW"` schaltet wieder zurück in den normalen Modus.

5.3.8 Linker

5.3.8.1 Library-Pfade: -j

Der Pfad, über den der Linker die Bibliotheken lädt, ist mit "*MCPP:lib*" voreingestellt und kann mit der Option `"-j"` verändert werden, z. B. `"-j dh1:cpp/lib"`. Genau wie bei der Option `"-i"` (5.2.3.2) kann auch hier ein mit `"|"` getrennter Doppelpfad wie z. B "*ram:clib|MCPP:lib*" angegeben werden, um die benutzten Dateien automatisch auf ein schnelleres Speichermedium umkopieren zu können. Im Gegensatz zu den Include-Pfaden kann für Libraries aber immer nur ein Pfad bzw. nur ein Doppelpfad definiert werden.

5.3.8.2 Linker-Libraries: -l

Um Linker-Bibliotheken (2.5.3.3) einbinden zu lassen, setzt man ihren Namen einfach hinter die Option `"-l"`, z. B. wird bei

```
mcppc ami.c -lamiga
```

die "*amiga.lib*" eingebunden. Hinter dem `"-l"` darf optional auch ein Leerzeichen stehen, und natürlich kann man auch mehrere Libraries einbinden.

5.3.9 Die integrierte Projektverwaltung: -m, -mx und -mX

Das Make-Utility gibt es nicht nur im der integrierten Umgebung "*MCPP*", sondern auch im *mcppc*. Man übersetzt anhand eines Makefiles, indem man die Option `"-m"` benutzt, z. B.

```
mcppc -m makefile
```

Die Make-Datei muß die in Abschnitt 3.3 beschriebene Form besitzen. Man kann zusätzliche globale Optionen für die Übersetzung angeben, z. B.

```
mcppc -m makefile -w200 -gO -gd
```

Mit diesen Einstellungen werden dann die C++-Objektdateien übersetzt, sofern in deren Compileranweisung nicht expilzit gegenteilige Optionen stehen. Auch Linkeroptionen können hier angegeben werden.

Namen zusätzlicher Text- und/oder Objektdateien werden im Make-Modus in der Kommandozeile naheliegenderweise nicht akzeptiert.

Die Pflege einer Projektdatei ist eine mühsame Angelegenheit. Wenn Sie keine Lust haben, die Abhängigkeiten von Hand zu setzen, oder fürchten, dabei etwas falsch zu machen, können Sie (zusätzlich zu `"-m"`) die Option `"-mX"` benutzen. Dann paßt *MCPPC* beim Compilieren auf, welche Quelltextdateien für welche Objektdatei benötigt werden, und aktualisiert anschließend das Makefile.

Aber Vorsicht: Ihre womöglich kunstvoll und elegant geschriebene Projektdatei wird dabei durch ein schnödes MCPP-Standard-Makefile geplättet. Insbesondere verschwinden dabei Kommentare, Makros und Regeln auf nimmerwiedersehen.

Wie Sie vielleicht schon vermutet haben, ist `"-mx"` der zu `"-mX"` passende Ausschalter.

Zum Schluß noch ein Tip: Sie können sich auch ein ganz neues Makefile erzeugen lassen, wenn Sie `"-mX"` ohne `"-m"` verwenden. Bei einem Aufruf wiem

```
mcppc x.c y.c z.c -mX
```

erzeugt *MCPPC* eine Projektdatei namens "*makefile*", in der die drei Objekte "*x.o*", "*y.o*" und "*z.o*" mit allen Compileranweisungen, Quelltexten und Anhängigkeiten stehen.

5.3.10 Arbeitsspeicher: -w

MaxonC++ alloziert zwar den Speicherplatz für die meisten internen Angelegeheiten automatisch und dynamisch, aber bei der Codegenerierung wird trotzdem ein zusammenhängender Speicherbereich benötigt, dessen Größe mit 80 KByte voreingestellt ist. Sollte bei einem umfangreichen Programm einmal eine Meldung wie **Workspace overflow!** ausgegeben werden, sollten Sie die Größe dieses Speicherbereichs mit der Option `"-w"` behutsam erhöhen, z. B. `"-w150"` für 150 KByte.

5.3.11 Vorcompilierte Headerdateien: -H, -h und -h0

Seit Version 3 kann MaxonC++ Programme vorcompilieren, d. h. an einer vom Benutzer frei wählbaren Stelle kann der Compiler alle seine internen Daten - und damit die bisher gelesenen Deklarationen und Definitionen - abspeichern. Beim nächsten Compilerdurchlauf kann diese Datei dann sehr schnell geladen und so viel Zeit gespart werden.

Dieses Feature ist dafür gedacht, die Programmentwicklung zu beschleunigen, wenn sehr viele Header-Dateien benutzt werden, z. B. die AmigaOS-Includes oder die MaxonC++ Klassenbibliothek "*EasyObjects*". Damit der Compiler aber weiß, wo die Header aufhören und Ihr eigenes Programm anfängt, muß man ihm diese Stelle mit einer Zeile **"#pragma header"** markieren.

Am besten schreiben Sie in den Quelltext jeder Übersetzungseinheit zunächst die **"#include's"** all jender Headerdateien, die Sie nicht selbst geschrieben haben oder an denen sich aus anderen Gründen nichts mehr ändern wird. Dann folgt die Zeile **"#pragma header"**, und anschließend folgen die Include-Anweisungen für Ihre eigenen Header-Dateien und der Rest Ihres Programms.

Ohne besondere Optionen, oder mit dem expliziten Ausschalter **"-h0"**, hat **"#pragma header"** keinerlei Wirkung. Wenn Sie aber eine Option **"-H <name>"** angeben, hält der Compiler, sobald das Pragma erreicht wird, kurz inne und schreibt alle Definitonen und Deklarationen in die Datei **"<name>"**. Danach arbeitet er weiter, als sei nichts geschehen.

Um das Programm anschließend beschleunigt erneut zu compilieren, wählen Sie eine Option wie **"-h <name>"**. Dann liest der Compiler die vorcompilierten Header aus der angegebenen Datei, sucht im Quelltext nach dem **"#pragma header"** und setzt dort mit seiner Übersetzungstätigkeit ein.

Da alles, was vor dem **"#pragma header"** steht, komplett ignoriert wird, muß es folglich in der Haupt-Quelltextdatei der Übersetzungseinheit - und nicht etwa in einer Includedatei - stehen.

Denken Sie bitte daran, daß *MCPPC* nicht prüft, ob der Inhalt der vorcompilierten Header-Dateien überhaupt noch mit dem ursprünglichen Quelltext übereinstimmt. Deshalb empfehle ich noch einmal dringend, nur solche Header-Dateien vorzucompilieren, an denen sich während der Programmentwicklung nichts mehr ändert.

5.4 Voreinstellungen mit "CPPOPT"

Wenn es Ihnen zu mühsam ist, bei jedem Compileraufruf Ihre ieblings-Optionen von Hand einzutippen, können Sie sich natürlich in einer geeigneten Shell ein passendes Alias definieren, z. B. haben Sie sich durch

```
alias cc2 "mcppc -pc -w300 "
```

einen C-Compiler mit großem Workspace geschaffen. Eleganter geht dies aber mit der Environment-Variablen **"CPPOPT"**, die freilich erst ab Betriebssystemversion 2.0 unterstützt wird:

```
setenv CPPOPT -pc -w300
```

stellt die beiden genanntem Optionen als Default ein. Die Variable wird so behandelt, als würde ihr Wert an den Anfang der Kommandozeilenargumente gestellt. Deshalb können Sie hier theoretisch nicht nur irgendwelche Optionen wählen, sondern z. B. auch mit

```
setenv CPPOPT test.c
```

zusätzliche Quelltexte oder Objektdateien angeben. Dies dürfte aber nicht unbedingt sinnvoll sein.

5.5 Zusammenfassung

Es folgt noch einmal eine kurze, alphabetisch geordnete Übersicht über alle CPPC-Optionen:

`-a`	erzeugt für jedes übersetzte Textfile einen Assemblerquelltext.
`-as`	C-Quelltext wird in Assembler-Source als Kommentar eingefügt.
`-a0`	Keine Assembler-Ausgabe (Default)
`-b`	Debug-Info-Datei erzeugen.
`-bi`	Wie `"-b"`, aber Icon für "*.mdbg*"-File erzeugen.
`-b0`	Ohne Debug-Info-Datei (Default)
`-bs`	"Symbol-Hunk" für symbolischen Debugger erzeugen.
`-bs0`	Kein Symbolhunk (Default)
`-c`	Die Quelltexte werden compiliert und die entsprechenden ".o"-Dateien erzeugt. Das Programm wird nicht gelinkt.
`-cx`	Wie `"-c"`, aber das ganze Projekt wird trotzdem gelinkt und eine ausführbare Programmdatei erzeugt.
`-cX`	Übersetzen und linken, aber keine Objektdateien schreiben (Default)
`-d NAME`	definiert ein Preprozessor-Symbol. Optional kann unmittelbar hinter den Namen ein `"="` und der gewünschte Wert (nur ein Token) gesetzt werden, andernfalls ist das Symbol als „leer" definiert.
`-eB`	Fehlermeldungen fett schreiben
`-eb`	Fehlermeldungen normal schreiben (Default)
`-ecN`	Bei Fehlermeldungen Quelltext in Farbe Nummer `"N"` ausgeben
`-ef xxx`	Fehlermeldungen in Datei "*xxx*" schreiben
`-ei`	Bei jeder Compiler-Fehlermeldung wird der Benutzer gefragt, ob abgebrochen werden soll (Default)
`-e[N]`	Fehlermeldungen werden ausgegeben, ohne daß Interaktion des Benutzers verlangt wird. Spätestens nach dem `"N"`-ten Fehler wird abgebrochen (Default ist 10).
`-el[N]`	Wie `"-e"`, aber zusätzlich mit Quelltextausgabe
`-ew`	Warnungen wie Fehler behandeln
`-eW`	Warnungen ausgeben und ansonsten ignorieren (Default)
`-gb`	Breakpoints setzen
`-gb0`	Keine Breakpoints setzen (Default)
`-gd`	Datenmodell "Small Data".

`-gD` Datenmodell "Large Data" (Default)

`-gh xxx` legt die maximale Größe (in KByte) fest, bis zu der gleichartige Hunks vom Linker zusammengefaßt werden. Default ist 50.

`-gl` Alle Funktionen in einer Section (Default)

`-gL` Library-Modus: Für alle Funktionen und jede globalen Variablen eigene Sections erzeugen

`-gm` MULS/DIVS-Operationen des Prozessors benutzen (Default: Bibliotheksfunktionen für korrekte 32-Bit-Arithmetik).

`-go` ohne Optimierungen (Default)

`-gO` mit Optimierungen

`-gs` Ohne Startup-Code linken

`-gsi` Wie `"-gs"`, aber Hunk mit `"InitModules"` und `"CleanupModules"` nur erzeugen, wenn diese Symbole auch irgendwo benutzt werden (praktisch, wenn man *MCPPC* als schlichten Linker z. B. für Assembler-Projekte mißbraucht).

`-gsw` Wie `"-gsi"`, aber zusätzlich Warnung ausgeben, wenn `"InitModules"` oder `"CleanupModules"` zwar nicht leer sind, aber trotzdem nirgendwo benutzt werden. Dann hat der Programmierer diese Aufrufe offenbar vergessen.

`-g00` Code für 68000 erzeugen (Default)

`-g20` Code für 68020 erzeugen

`-g30` Code für 68030 erzeugen

`-g40` Code für 68040 erzeugen

`-g80` Code ohne FPU-Benutzung erzeugen (Default)

`-g81` Code für 68881-FPU erzeugen

`-g82` Code für 68882-FPU erzeugen

`-H xxx` vorcompilierte Header-Dateien abspeichern

`-h xxx` vorcompilierte Header-Dateien laden

`-h0` Ohne vorcompilierte Header übersetzen (Default)

`-i xxx` Den Pfad bzw. die Pfade, über den *MCPPC* die Includefiles sucht, setzen. Mehrfaches `"-i"` für mehrere Pfade ist möglich. Defauli ist "*MCPP:include*". Tritt in einem Dateinamen ein `"|"` auf, werden die gelesenen Files automatisch vom zweiten in den ersten Pfad umkopiert und dabei kompaktiert, z. B. "*-i ram:include|c++:include*"

`-j xxx` definiert analog zu `"-i"` den Pfad bzw. den Doppelpfad, über den die Libraries gesucht werden. Hier kann aber nur ein Pfad bzw. nur ein Doppelpfad (mit `"|"`) angegeben werden. Default ist "*MCPP:lib*".

`-l xxx` Die Library "*xxx.lib*" wird aus dem Libs-Verzeichnis eingebunden.

`-L` Abkürzung für `"-gL"`

`-m xxx` übersetzt nach Makefile "*xxx*".

`-mX` Abhängigkeiten in Makefile automatisch setzen

`-mx` Makefile bloß lesen (Default)

`-o xxx` legt einen anderen Namen für die Exe- bzw. die Objektdatei fest.

`-O` Abkürzung für `"-gO"`

`-pc` schaltet in ANSI C-Modus
`-pp` C++-Modus (Default)
`-px` Ohne Exception Handling
`-pX` Exception Handling ermöglichen
`-t` Nur fehlerfreie Template-Funktionen erzeugen
`-T` Alle benötigten Template-Funktionen erzeugen
`-w xxx` Warnings ausgeben. Es kann eine beliebige Folge von Groß- und Kleinbuchstaben folgen, wobei ein GROSSER Buchstabe die Warnung einschaltet.

Folgende Warnungen gibt es:

`a` (Assignment): Operator `"="` taucht in Bedingung auf - vielleicht der bekannte `"=="`-Anfänger-Fehler?
`c` (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. `double -> int`
`e` (Empty Statement): Anweisung hat keinen Code, z. B. `"42;"`
`f` (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)
`n` (Nested comments): `"/*"` in Kommentar
`m` (Macro): Ein Argument für ein Preprozessor-Makro erstreckt sich über mehr als 8 Zeilen oder 200 Token. Es liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.
`o` (Old-Style): Über Parameterlisten im alten "K&R"-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)
`p` (Pragma): Unbekanntes `"#pragma"`
`r` (Return): Funktion mit Rückgabewert hat kein `"return"`.
`t` (Temporary): Temporäres Objek wurde eingeführt, um ein nicht-`"const"`-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.
`v` (Variable): Lokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind `"-wEPRT"` aktiviert.

`-wN` Setzt den compilerinternen Workspace auf "N" KByte. Default sind 80 KByte, und das sollte reichen.
`-w+` Alle Warnungen ausgeben
`-w` Keine Warnungen ausgeben
`-X` Abkürzung für `"-pX"`

Anhang A

Veränderungen seit der Version 1.0

Seit der Version 1.0 hat sich bei Maxon C++ offensichtlich einiges getan. Die diversen Bug-Fixes möchten wir hier nicht aufführen, denn meist ist die exakte Beschreibung der Fehler dermaßen kompliziert, daß sich kaum jemand das alles durchlesen würde. Auch viele Details in der Code-Erzeugung wurden verbessert, was der Programmierer aber nicht unbedingt wissen muß, denn das äußert sich ausschließlich in teilweise kompakteren und schnelleren Programmen.

Die "großen" Erweiterungen und Veränderungen, etwa die Exceptions und Templates oder die völlig überarbeitete Entwicklungsumgebung, dürften Ihnen schon aus dem Vorwort des Benutzerhandbuchs oder durch bloßen Augenschein bekannt sein. Deshalb beschränken wir uns hier auf einen Hinweis auf Abschnitt 5.1 des Benutzerhandbuchs, wo die Änderungen an der CLI-Compilerversion "*mcppc*" dargestellt werden, und gehen gleich zu den vielen kleinen, aber feinen Änderungen über:

A-1. Compiler

A-1.1 Preprozessor

- **"Pseudo-Vorcompilieren" von Includes:**
  Beim Umkopieren von Includes werden diese vorgescannt - bringt ein bißchen 'was an Compilezeit und spart enorm Speicher, wenn in die RAM-Disk umkopiert wird.
- Die interne Compiler-Hashtable ist größer und kostet trotzdem weniger Speicher.
- Makro `"__DATE2__"` liefert das Datum in vernünftigem Format.
- Makro `"__MAXON__"` wurde auf `"30"` geändert.
- Das `"#pragma header"` wurde zwecks Vorcompilieren von Header-Dateien eingeführt.

A-1.2 Parser und Sprachstandard

- Pointer auf Member-Funktionen sind jetzt implementiert.
- Beim bedingten Ausdruck (`c ? a : b`) mit konstanter Bedingung `"c"` wird die Fallunterscheidung wegoptimiert, und der ganze Ausdruck ist dann u. U. konstant (letzteres ist strenggenommen ein Bug-Fix)
- Bei der Pointer-Subtraktion kann jetzt die Differenz zwischen einen `"T *"` und einen `"const T *"` berechnet werden. Was hier der Standard vorschreibt, ist nicht ganz klar, aber die neue Regelung ist sinnvoll und harmlos.
- Operatoren `"+="`, `"|="` etc. funktionieren jetzt auch auf Bitfeldern.

- Neue Fehlermeldung, wenn im Small Data Model ein Parameter in Register A4 übergeben wird, und in Large Data ist das jetzt fehlerfrei möglich.
- Warnung `#10` für „Verdächtiges" `'='` in Bedingung wird nicht mehr ausgegeben, wenn die Zuweisung in Klammern steht (z. B. `"if ((a=b))..."`)
- In `"int main()"` wird bei fehlendem `"return"` nicht mehr gewarnt.
- Sinnreichere Fehlermeldung `"Class/struct xxx undefined"` bei Memberzugriff auf undefinierte Struktur
- Bei abstrakten Klassen wird ggf. der Name der fehlerhaften Basisklasse in der Fehlermeldung ausgegeben (Fehler `#179` und `#181`).
- Die Schlüsselworte `"catch"`, `"throw"` und `"try"` sind gesperrt, wenn das Exception-Handling nicht per Compiler-Option eingeschaltet wurde.

A-1.3 Codeerzeugung

- Datentyp `"float"` wurde von "FFP" auf "IEEE-Single" umgestellt.
- Im "Library-Modus" werden alle Funktionen und globale Variablen in separate Hunks gepackt.
- Alignment bei mehrdimensionalen Vektoren ist besser gelöst, z. B. ist `"char x[10][9]"` jetzt 90 (bisher: 100) Bytes groß.
- 68020/30-Code mit Option `"-g20"` bzw. `"-g30"` (benutzt vorerst nur Langwort- Multiplikation und -Division sowie Skalierung bei Index-Adressierung)
- Direkte FPU-Unterstützung
- Bedingungstest bei `"while"`-Schleifen steht jetzt am Schleifenende (spart einen Branch pro Durchlauf).
- Inlining von Funktionen geschieht auch, wenn Funktionsdefinition hinter Funktionsaufruf steht
- Identische Zeichenketten werden nur noch einmal abgespeichert
- Längst überfällige Codeverbesserungen, z. B. wird `'*p++'` ggf. in `'(aN)+'` übersetzt

A-1.4 Linker

- Alle Varianten von `"main"` werden ins "*.mdbg*"-File eingetragen (`"main_"`, `"main_iPPc"`, `"_main"` und `"_wbmain"`).
- Bei Linkerfehlern werden die Linkernamen nach Möglichkeit in C++-Funktionsdeklarationen rückübersetzt

☞ Jetzt sind bis zu 1000 Hunks pro geladener Objektdatei möglich (früher nur 100)

A-2. Bibliotheken

☞ Stubs für ASL-, DOS- und Intuition-Tag-Funktionen (erspart amiga.lib)

☞ Die Funktion **"wbparse"** erleichtert den Workbench-Startup, insbesondere im Zusammenhang mit dem Include *<wbstartup.h>*.

☞ Die *"ctype.h"*-Funktionen sind um eine Winzigkeit schneller und kompakter geworden.

☞ Die Cleanup-Priorität 0 (**"_EXIT_0_..."**) wird nur noch fürs Beantworten der Startup-Message (bei Workbench-Startup) benutzt. Wer Assemblermodule schreibt, die eine solche Cleanup-Funktion besitzen, sollte die Priorität 0 deshalb nicht mehr benutzen und bedenken, daß alle anderen Library-Module jetzt ihr Resource Tracking o. Ä. auf Level 1 und 2 durchführen.

☞ Die Funktionen **"memmove"** und **"memcpy"** sind jetzt identisch, kommen also beide mit überlappenden Speicherbereichen klar. Außerdem wurden sie durch langwortweises Kopieren (falls möglich) beschleunigt.

☞ **"realloc"** arbeitet jetzt Amiga-konform (siehe auch Abschnitt 1.1.4 im Referenzhandbuch)

A-3. Sonstiges

☞ Defaults für diverse Pfade und Dateinamen wurden vereinheitlicht (z. B. logisches Verzeichnis *"MCPP:"* und alle Executables in *"MCPP:bin/"*)

☞ Auf vielfältigen Wunsch setzt **"-o"** in Verbindung mit **"-c"** jetzt auch den Namen der Objekt-Datei. Werden mehrere Objektdateien auf einmal übersetzt, bezieht **"-o"** sich ausschließlich auf den erstgenannten Quelltext.

☞ Der Chunk **"HUNK"** wird in den *".mdbi"*-Dateien nicht mehr geschrieben (war überflüssig und wäre durch die massig vielen möglichen Hunks etwas aufwendig gewesen)

☞ *MCPP* definiert für Edward jetzt das Symbol **"MCPP3"** (wg. *DEF-Datei*).

Anhang B – Compiler Fehlermeldungen

MaxonC++ kennt etwa 250 Fehlermeldungen und ein gutes Dutzend Warnungen. In der vorliegenden Version sind diese noch ausschließlich Englisch, für die nähere Zukunft ist aber vorgesehen, sie zu lokalisieren. Deshalb werden sie hier nach Nummern sortiert. Beachten Sie bitte, daß Fehler und Warnungen getrennt numeriert sind.

B-1. Warnungen

MaxonC++ bietet dreizehn Warnungen, die größtenteils einzeln an- und abschaltbar sind:

1: Nested comment

Die Zeichenfolge `"/*"` innerhalb eines Kommentars ist verdächtig, da hier möglicherweise schlicht vergessen wurde, einen vorhergehenden Kommentar mit `"*/"` zu beenden.

2: No prototype for function "<name>".

Eine Funktion ohne Prototyp wurde aufgerufen. In ANSI C ist es zwar strenggenommen nicht erlaubt, Funktionen ohne Prototyp aufzurufen, aber aus Gründen der Kompatiblität mit dem hoffnungslos veralteten und ebenso hoffnungslos unausrottbaren "K&R"-Standard gibt es dafür im C-Modus nur eine abschaltbare Warnung. In C++ werden Sie diese Warnung nicht erhalten, denn da ist das ein echter Fehler.

3: Function "<name>" has no "return" statement.

Wenn in einer Funktion, deren Ergebnistyp nicht `"void"` ist, keinerlei `"return"`-Anweisung auftaucht, wurde diese offensichtlich vergessen.

4: Statement has no effect at all.

Es wurde eine Anweisung wie `"42;"` gefunden, die zwar syntaktisch absolut korrekt, aber offensichtlich sinnlos und deshalb wahrscheinlich nicht so beabsichtigt ist. Besonders häufig dürfte diese Warnung wohl beim Aufruf einer parameterlosen Funktion ohne Argumentliste (z. B. `"f;"`) auftreten.

5: Variable <xxx> declared but never used.

Natürlich dürfen Sie so viele Variablen deklarieren, wie Sie wollen, und niemand zwingt Sie, diese auch zu benutzen. Eine entsprechende Warnung, z. B. bei einer Variablen, die einmal benutzt worden war, inzwischen aber redundant ist und nur Speicherplatz kostet, ist aber sicher bisweilen ganz angenehm.

Es wird hier aber ausschließlich bei lokalen Variablen von Funktionen gewarnt, da der Compiler nicht übersehen kann, ob eine globale statische Variable vielleicht in einer anderen Übersetzungseinheit benutzt wird. Funktionsparameter werden in dieser Überprüfung ebenfalls nicht berücksichtigt.

6: This seems to be an old-style parameter list.

Der Compiler hat eine Parameterliste im alten Kernighan-Ritchie-Stil gefunden, oder glaubt es zumindest (es könnte auch ein beliebiger anderer Syntaxfehler sein). Diese Parameterlisten sind nicht schön, weshalb der Compiler bei Bedarf mittels einer Warnung seine Verachtung darüber zum Ausdruck bringen kann.

7: Unknown #pragma (maybe misspelled?)

Ein unbekanntes Pragma wurde gefunden - möglicherweise verdächtig.

8: Temporary object for non-const reference introduced.

Eine Referenz auf einen nicht konstanten Datentyp wurde initialisiert, wobei aber ein temporäres Objekt eingeführt werden mußte. Dies ist seit dem C++ 2.0-Standard strengenommen nicht erlaubt und außerdem immer eine suspekte Konstruktion.

9: Unsafe type conversion

In der vorliegenden Compiler-Version wird diese Warnung nur ausgegeben, wenn ein Fließkommawert ohne explizite Typkonvertierung (also ohne Cast) in einen ganzzahligen Typen umgewandelt wurde. Weitere "unsichere" Umwandlungen werden mit dieser Warnung eventuell in spätern Compilerversionen gemeldet.

10:Suspect "=" in condition.

Eine Wertzuweisung wurde als logische Bedingung benutzt, z. B. wie in `"if (a=b)"`. Das ist zwar allgemein beliebt, aber möglicherweise auch ein Versehen.

Wenn solche Konstrukte zu Ihrem persönlichen Programmierstil gehören, sie aber trotzdem deratigen Fehlern vorbeugen möchten, können Sie die Warnung überlisten: Durch eine zusätzliche Klammer, also z. B. `"if ((a=b))"`, an den Stellen, wo Sie es wirklich so meinen, wird die Warnung unterdrückt.

11:Precompiled header file "<name>" not found.

Vorcompilierte Headerdateien können ganz schön groß werden, weshalb man sie beim Aufräumen der Festplatte gerne wegschmeißt. Deshalb macht Maxon C++ keinen großen Aufstand, eine Headerdatei, die gelesen werden soll, einmal nicht vorhanden ist: Die Warnung Nummer 11 wird ausgegeben und das Programm anstandslos übersetzt, nur eben ohne die Header-Datei.

12:Variable <name> never initialized.

Eine lokale Variable wurde deklariert und benutzt, aber nie initialisiert, was offensichtlich ein Fehler ist. Der Compiler ist an dieser Stelle aber ziemlich leicht hinters Licht zu führen: Wenn irgendwo die Adresse der Variablen genommen wird, verfolgt er das nicht mehr weiter und geht einfach davon aus, daß die Variable irgendwie und irgendwann über diesen Zeiger einen Wert erhält. Da bei jedem Vektor-Zugriff implizit der Adressoperator auf den Vektor angewamdt wird, entfällt diese Warnung bei Vektoren. Außerdem ist der Compiler nicht schlau genug, durchgeknallte Anweisungen wie `"i &= 0;"` oder `"j -= j;"` als Initialisierung zu erkennen - aber wer macht auch schon sowas?

13: Very long argument for Macro "<xxx>" - probably missing ")".

Ein Argument für ein Makro erstreckt sich über mahr als 8 Zeilen oder 200 Token. Deshalb liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.

B-2. Fehlermeldungen

1: Memory overflow!

Der Speicher ist voll. Die Fehlermeldung bezieht sich NICHT auf den Compiler-Workspace (der kann frühestens während der Codegenerierung überlaufen), sondern auf den freien Systemspeicher. Abhilfe: So viel wie möglich RAM freimachen, oder eine Speichererweiterung kaufen...

2: String constant exceeds line.

Die Zeichenkette geht über das Zeilenende hinaus, oder die abschließenden Anführungszeichen wurden vergessen.

3: Char constant: unmatched '.

Ein einfaches Hochkomma `"'"` leitet eine Zeichenkonstante ein und kann deshalb immer nur paarweise auftreten.

4: Preprocessor command expected.

Hinter einem `"#"` wird eine Preprozessor-Anweisung stehen.

5: Identifier expected.

Hier wird ein Bezeichner erwartet. Im Gegensatz zur wortgleichen Meldung 60 tritt diese im Zusammenhang mit dem Preprozessor auf, z. B. bei einer fehlerhaften Makro-Definition.

6: Macro "<name>" redefined.

Das Preprozessor-Makro `<name>` wurde schon einmal anders definiert.

7: ")" expected.

Ein Preprozessor-Fehler: die abschließende Klammer bei Makrodefinition mit Parameterliste fehlt.

8: "##" at end of Macro.

Der Operator `"##"` verbindet innerhalb von Makrodefinitionen zwei Symbole und kann deshalb niemals am Ende des Makros stehen.

9: Macro expansion: "(" expected.

Das Makro erwartet eine Parameterliste.

10:"," expected.

Hier fehlt ein Komma, oder zumindest glaubt der Compiler das.

12:Compiler is in Panic!

Compiler-intern ist etwas höllisch schiefgelaufen! MaxonC++ ist ziemlich mißtrauisch programmiert und überprüft hin und wieder sich selbst. Wird dabei festgestellt, daß eine Situation vorliegt, die eigentlich unmöglich ist, steigt der Compiler mit dieser Fehlermeldung aus. Es handelt sich hier um einen Bug des Compilers, nicht um einen Fehler Ihres Programms! Wenn dieser Fehler auftritt, melden Sie sich bitte umgehend bei MAXON - am besten wäre, wenn Sie dabei möglichst präzise schildern könnten, wo und unter welchen Umständen der Fehler auftrat.

Als Notbehelf können Sie versuchen, das Programm an der bemängleten Stelle anders zu formulieren. Wahrscheinlich (hoffentlich) gerät der Compiler aber nur in Folge zuvor gemeldeter Fehlern in Panik.

13:Macro expansion: Illegal use of "#" or "##".

Die Token-Operatoren `"#"` und `"##"` werden im vorliegenden Makro falsch angewendet (Tutorial 5.2.2).

14:Recursive definition of macro.

Das Makro taucht direkt oder indirekt in der eigenen Definition auf.

15:Macro name expected.

Hier wird ein gültiger Makroname, also ein Bezeichner, erwartet.

16:"Garbage" at end of preprocessor command.

Die Preprozessoranweisung scheint vollständig zu sein, aber dahinter steht in der gleichen Zeile noch etwas nicht näher Identifizierbares.

17:Filename expected.

Hinter `"#include"` oder `"#line"` wird hier ein gültiger Dateiname erwartet.

18:Can not open file "<name>".

Der Preprozessor findet eine Includedatei nicht. Sie sollten im Zweifelsfall insbesondere überprüfen, ob Sie den Include-Pfad sinnreich definiert haben.

19:Can not copy file "<source>" to "<dest>".

Der Preprozessor hat irgendwelche Probleme, eine Include-Datei umzukopieren. Das kann die unterschiedlichsten, aber meist offensichtlichen Ursachen haben.

20:Overflow at nesting of Includes (max. depth is 25)

Der Preprozessor kann Includedateien höchstens 25-fach ineinander verschachteln (Änderung dieses Limits in weiteren Versionen vorbehalten). Bei einer derartigen Schachtelung liegt ohnehin der Verdacht nahe, daß Sie eine Art endlose Rekursion programmiert haben.

21:Can not open math library.

Für interne Fließkommaberechnungen, z. B. um Ausdrücke wie `"2*3.14159"` schon zur Compile-Zeit auswerten zu können, benötigt der Compiler die "mathieeedoubbas.library".

22:No matching "#if".

Der Preprozessor stieß auf ein `"#endif"` oder `"#else"`, für das es kein `"#if"` gibt.

23:At least one "#endif" is missing.

Am Ende des Quelltexts stellt der Preprozessor fest, daß wenigstens ein `"#if - #endif"`-Preprozessor-Konstrukt noch nicht abgeschlossen ist. Ich wünsche viel Spaß bei der Suche nach dem `"#if"`...

24:Error in condition.

In der Bedingung eines Preprozessor-`"#if"` wurde ein syntaktischer Fehler gefunden.

25:User Error: ...

Diese Fehlermeldungen werden durch die Preprozessor-Anweisung `"#error"` gezielt provoziert.

26:Line number expected.

In der Preprozessoranweisung `"#line"` wird als erster Parameter eine ganzzahlige Zeilennummer erwartet.

27:Pseudo-filename expected.

Als optionaler zweiter Parameter von `"#line"` wird ein Dateiname in Form eines Strings erwartet.

29:Compiler stack overflow!

Der Stack des Compilers ist übergelaufen. Am besten versuchen Sie, *MCPP* bzw. *mcppc* nach Heraufsetzen des Stacks erneut zu starten. Beachten Sie bitte, daß dies nichts(!) mit der Eintragung "Laufzeit-Stack" in der System-Dialogbox von *MCPP* zu tun hat.

Wenn Sie den mcppc benutzen oder die Entwicklungsumgebung *MCPP* von der Shell aus gestartet haben, können Sie mit der Shell-Anweisung "stack" zuerst die aktuell gültige Stackgröße feststellen und dann einen geeigneten höheren Wert setzen. 10000 Bytes sind normalerweise genug, und mit 20000 sind Sie klar auf der sicheren Seite.

Falls Sie aber *MCPP* von der Workbench aus durch Anwählen und Öffnen eines Icons gestartet haben, sollten Sie auf dieses Icon den Workbench-Menüpunkt **"Information"** anwenden. In der Dialogbox, die sich dann öffnet, können Sie die gewünschte Stackgröße eintragen.

30:Too many errors.

Spätestens nach 50 Fehlern weigert sich MaxonC++ weiterzumachen. Warnungen werden dabei normalerweise nicht mitgezählt.

35:Declarations are not allowed here.

In C++ gelten Deklarationen zwar auch als Anweisungen, hier aber (z. B. direkt und ohne umschließende Blockklammern hinter `"if"` oder `"else"`) wird eine Anweisung, die keine Deklaration ist, erwartet.

36:Unions may not have members with Constructors or Destructors

Member von Unionen dürfen keine Konstruktoren oder Destruktoren besitzen (Tutorial 2.5.5 Ende).

37:A global anonymous union must be declared "static".

Deklariert man eine namenlose globale Union, also mehrere statische Objekte mit identischem Speicher, muß diese aus Gründen der Konsistenz und um den Linker nicht unnötig zu verwirren mit der Speicherklasse `"static"` geschehen. So ist nun mal die Regel, tut mir leid.

38:Functions can not be defined local to functions

Funktionsdefinitionen können in C (im Gegensatz z. B. zu Pascal) nicht ineinander verschachtelt werden. Vielleicht haben Sie bei der vorherigen Funktionsdefinition auch nur die schließende Klammer vergessen.

39:Function parameters differ from first declaration.

Da es in C kein Überladen von Funktionen gibt, kann der Compiler mit diesen widersprüchlichen Angaben über die Paraleterliste nicht leben. Im C++-Modus werden Sie dieser Fehlermeldung nicht begegnen.

40:";" expected.

Ein Semikolon `";"` fehlt - eine Fehlermeldung, die so ziemlich überall auftreten kann.

41:Incomplete type-specification.

Der Datentyp ist nicht hinreichend definiert, insbesondere ist z. B. seine Größe unbekannt. Wahrscheinlich versuchen Sie an dieser Stelle gerade, ein Objekt einer noch nicht definierten Klasse oder Struct zu definieren.

42:Vectors of references, functions or void are illegal.

Referenzen, Funktionen und `"void"` können nicht Elementtyp eines Vektors sein.

43:Vector has element type of unspecified size.

Der Datentyp der Vektorelemente (und damit auch der Vektortyp selbst) hat einen Speicherplatzbedarf, der dem Compiler momentan nicht bekannt ist. Möglicherweise habdelt es sich hier um eine bisher undefinierte Klasse.

44:Pointers/references to references are illegal.

Zeiger oder Referenzen auf Referenzen, also z. B. `"int &&"` oder `"int *&"`, sind nicht erlaubt.

45: Illegal referenced type.

Referenzen auf Funktionstypen und ähnlich abgefahrene Dinge gibt's nicht.

46: Function has illegal return type.

Der Ergebnistyp der Funktion ist ungeeignet, z. B. ein Vektor oder eine Funktion.

47: Declaration expected.

Hier erwartet der Compiler irgendeine Deklaration.

48: Struct/class "<xxx>" undefined.

Die angegebene Klasse ist zwar dem Namen nach bekannt (also deklariert), aber nicht definiert.

49: ")" expected.

An dieser Stelle würde der Compiler gern eine schließende Klammer sehen.

50: Identifier "<name>" redefined.

Der Bezeichner wurde im selben Gültigkeitsbereich schon einmal definiert, und zwar anders.

51: Definition has illegal Type.

Variablen und Member dürfen als Typ nicht `"void"`, eine undefinierte Klasse oder etwas ähnlich Sinnloses haben.

52: Syntax error in type description.

Ein Syntax-Fehler wurde in einem Konstrukt entdeckt, das der Compiler für eine Typbeschreibung hält.

53: Please switch to C-Mode for old style declarations.

Parameterlisten im alten "K&R"-Stil erlaubt MaxonC++ nur im ANSI C-Modus.

54: No actual "this" pointer.

Die Variable `"this"` gibt es hier nicht. Man kann sie nur in nicht-statischen Memberfunktionen benutzen.

55: Friend declaration: "<xxx>" unknown.

Wird in einer `"friend"`-Deklaration ein qualifizierter Bezeichner benutzt, z. B `"friend int C::f()"`, muß dieser Bezeichner (insbesondere diese Funktion) bereits deklariert sein, und zwar exakt typgleich.

56: This overloaded function has different access modes.

Zugriffs-Deklarationen sind nur für solche überladene Funktionsnamen erlaubt, bei denen sämtliche Funktionen gleichen Zugriffsmodus haben (Tutorial 3.2.2.3 Ende).

57: Member function re-declared.

Die Memberfunktion wurde nach Ende der Klassendefinition erstmals oder anders als bisher definiert.

58: Tried to change functions access.

Bei der Deklaration einer Member-Funktion wurden widersprüchliche Angaben über den Zugriffsmodus gemacht.

59: Return type differs form first declaration.

Bei einer vorhergehenden Funktionsdeklaration wurde ein anderer Ergebnistyp angegeben. Könnte auch sein, daß sie zuvor versehentlich ohne Prototyp aufgerufen wurde, der Compiler einen solchen mit Default-Ergebnistyp `"int"` eingeführt hat und nun ein anderer Typ deklariert wird.

60: Identifier expected.

Hier wird ein Bezeichner erwartet. Im Gegensatz zur wortgleichen Fehlermeldung Nummer 5 wird diese vom Parserteil des Compilers ausgeworfen.

62: Syntax error in enumeration.

Ein Syntx-Fehler in der Definition eines Aufzählungtyps.

63: Unions may not be derived.

Unionen dürfen keine Basisklassen haben. Könnte aber auch sein, daß der Doppelpunkt hinter dem Unions-Namen einfach ein Versehen ist.

64: ":" expected.

Das Symbol `":"` fehlt hinter Schlüsselworten wie `"private"` oder `"public"` auf.

65: No access to member "<name1>" of class "<name2>".

Auf den Member oder sonstigen Klasseninternen Bezeichner `<name1>` der Klasse `<name2>` darf hier nicht zugegriffen werden.

66: Can not define qualified identifier.

Nach der Definition einer Klasse wurde scheinbar versucht, einen bisher unbekannten Member bzw. eine Memberfunktion hinzuzufügen, was natürlich nicht statthaft ist. In der Regel haben wir es hier aber bloß mit einem Schreibfehler zu tun.

67: Data type undefined.

Die Größe dieses Datentyps ist bisher nicht bekannt, weshalb er hier nicht in dieser Weise benutzt werden kann.

68: Illegal use of storage class specifier.

Die angegebene Speicherklasse (z. B. `"auto"`) ist hier (z. B. in einer Klassen- bzw. Strukturdefinition oder auf Dateiebene) sinnfrei.

69: Class members can not get initialized.

Member von Klassen, auch nicht statische, dürfen innerhalb der Klassendefinition nicht initialisiert werden.

70: Syntax error in base-list.

In der Basisklassenliste einer Klasse wurde ein Fehler (das "syntaktisch" ist hier nicht so ernst zu nehmen, es kann auch ein semantischer sein) gefunden.

71: Multiple use of same base class.

Jede Klasse kann jede direkte Basisklasse nur einmal haben. Die bemäkelte Klasse ist in der Basisklassenliste aber schon einmal aufgetaucht.

72: Illegal use of "friend" specifier.

Hier (z. B. vor einer Variablen- oder Typdefinition) ist keine `"friend"`-Spezifizierung erlaubt.

73: Illegal use of Bitfields.

Mischungen von Bitfeldern und gewöhnlichen Membern sind in MaxonC++ nicht erlaubt (Tutorial 2.5.6).

74: Attempt to change access of member "<xxx>" of class "<yyy>".

Bei einer Zugriffs-Deklaration (Tutorial 3.2.2.3) wurde versucht, den Zugriffsmodus eines Members zu verändern.

75: Anonymous union: Illegal member "<xxx>".

Eine namenlose Union darf keine Member-Funktionen, statische Member und dergleichen haben, was auch irgendwie logisch ist (Tutorial 2.5.5 Ende).

76: Register objects must be local to a function.

Die Speicherklasse `"register"` ist nur innerhalb von Funktionen zulässig.

77: Syntax error in declaration.

Auf vielfältige Weise kann hier in dem, was der Compiler für eine Deklaration hält, ein falscher Fehler stecken.

78: Access to variable that was constructed under condition.

Tritt eine Variablendeklaration in einer Bedingung oder als Schleifenrumpf auf, ohne mit `{ ... }` geklammert zu sein, ist sie außerhalb des Schleifen- oder Abfragekonstrukts zwar prinzipiell sichtbar, darf aber nicht benutzt werden. Beispiel: Bei der Anweisung `"while (i) int j = i++;"` darf `"j"` später nicht mehr benutzt werden.

79: Syntax error in expression or declarator.

Wegen der konzeptionellen Ähnlichkeit von bestimmten Ausdrücken und Deklarationen, und weil in C++ manchmal erst in einer sehr späten Phase der Analyse zwischen Ausdrücken und Deklarationen unterschieden werden kann, wird deren Syntax teilweise schon geprüft, wenn der Compiler noch nicht weiß, was er vor sich hat. Immerhin hat er schon zu diesem Zeitpunkt feststellen können, daß es sich um einen syntaktischen Fehler handelt.

80:Constant integer expression expected.

Hier wird (z. B. als Elementanzahl eines Vektors) ein konstanter ganzzahliger Ausdruck erwartet.

81:"]" expected.

Die schließende Klammer "]" zu einer entsprechenden öffnenden Klammer fehlt - das könnte aber auch Folgefehler eines anderen Syntax-Errors sein.

82:Initialized object is not an aggregate.

Das Objekt, das hier mit einer Elementliste "{ ... }" initialisiert werden soll, ist kein Aggregat, also kein Vektor (Tutorial 2.1.2.6) und keine geeignete Struktur, die keine Basisklassen, Konstruktoren oder private Memberbesitzt (Tutorial 2.5.3).

83:"}" expected.

Die schließende Klammer "}" zu einer entsprechenden öffnenden Klammer fehlt - das könnte aber auch Folgefehler eines anderen Syntax-Errors sein.

84:Too many Vector elements given

Bei einer Listeninitialisierung wurden mehr Werte angegeben als der Vektor Elemente hat.

85:Constant String is too long

Ein Zeichenvektor wurde mit einem unzulässig langen String initialisiert. Beachten Sie hier auch den kleinen Unterschied zwischen C und C++ (Tutorial 2.1.2.5.2).

86:Initializer has illegal Type.

Der Initialisierungswert paßt vom Typ her nicht.

87:Too many struct/class members.

Bei einer Listeninitialisierung wurden mehr Werte angegeben als die Struktur Member hat.

88:Bitfield member has illegal size

Ein Member eines Bitfelds darf maximal 32 Bit breit sein, und natürlich muß die Bit-Breite auch eine echt positive Zahl sein.

89:Identifier "<xxx>" not defined.

Der angegebene Bezeichner ist bisher unbekannt. Es wird angenommen, daß es sich dabei um eine Variable handelt.

90:Member of Class "<xxx>" expected.

Hier, z. B. hinter "." oder "->", wird ein Member der genannten Klasse erwartet.

91:Access to ambiguous base class

Es wurde auf einen nicht-statischen Member oder eine nicht-statische Memberfunktion zugegriffen, die im Ableitungsbaum der vorliegenden Klasse mehrfach auftritt, wodurch dem Ausdruck kein eindeutiges Objekt zugeordnet werden kann.

92: Function call: can not use register A4 in small data model.

Das Register A4 wird bei Übersetzung im Small Data-Modell dringend anderweitig benötigt und kann deshalb nicht zur Parameterübergabe an Funktionen benutzt werden.

93: Subscribt: int expression in []'s expected.

Der Indexausdruck beim Vektorzugriff muß ganzzahlig sein.

94: Subscript: operand before []'s must be vector or pointer.

Vor dem mit "[ ]" geklammerten Index muß ein Zeiger- oder Vektorausdruck stehen.

96: Illegal type conversion

Diese Typumwandlung ist noch nicht einmal durch einen Cast möglich.

97: Operators "++" and "--" need a l-value.

Der Operand der unären Operatoren "++" und "--" muß ein L-Wert sein (Tutorial 1.4.3.2, 1.4.3.3).

98: Unary operator used on illegal type.

Ein unärer Operator wurde auf einen absolut unpassenden Ausdruck angewendet.

100: Address operator "&" needs a l-value.

Der unäre "&"-Operator kann ausschließlich auf L-Werte (also solche Ausdrücke, die z. B. auch links vom Zuweisungsoperator "=" stehen können) angewandt werden. Bitfeld-Member sind in diesem Zusammenhang keine L-Werte.

101: Function call: Expression left of () is not a function.

Der Compiler hält ein Klammerpaar "( )" (das nicht notwendig leer sein muß) für eine Argumentliste, den Ausdruck links davon aber nicht für eine Funktion.

103: Assignment: Type conflict.

Nur eine aufgeblähte Verrsion der Fehlermeldung #128 ("Type conflict"), die anzeigt, daß die nicht zueinander passenden Datentypen diesmal auf den beiden Seiten einer Wertzuweisung ("=") stehen.

104: Syntax error in Parameter list

In der Parameterlisten-Deklaration steckt ein syntaktischer Fehler.

105: Parameter has illegal Type.

Der Typ dieses Funktionsparameters ist unzulässig, z. B. könnte er bisher nicht hinreichend definiert sein.

106: Unknown function "<xxx>".

Zur dieser Funktion existiert kein Prototyp.

107: Default parameter has illegal type.

Der Typ des hier angegebenen Default-Arguments paßt nicht zum Parameter.

108: Double use of parameter register.

Das hier explizit angegebene Register ist in der vorliegenden Parameterliste schon anderweitig vergeben.

109: Redefinition of default argument for parameter "<xxx>".

Default-Argumente dürfen jeweils nur genau einmal spezifiziert werden (Tutorial 1.6.5.2).

110: Default Argument not at end.

Default-Argumente dürfen nur für die letzten Parameter einer Funktion definiert werden, d. h. es darf kein Parameter ohne Defaultargument mehr folgen (Tutorial 1.6.5.2).

111: Redefinition of default argument.

Default-Argumente dürfen jeweils nur genau einmal spezifiziert werden (Tutorial 1.6.5.2).

112: Argument missing.

Beim Funktionsaufruf wurde mindestens ein Argument zu wenig angegeben.

113: Ambiguous function call.

Beim vorliegenden Aufruf eines überladenen Funktionsnamens lassen die Argumente keinen eindeutigen Schluß auf die gewünschte Funktion zu (Tutorial 4.1.2).

114: Illegal argument.

Dieses Funktionsargument hat einen unpassenden Datentypen.

115: Function declaration: Illegal storage class.

Eine Funktion darf als Speicherklasse zwar `"extern"` oder `"static"`, nicht aber `"auto"`, `"register"` oder sonstwas Bescheuertes besitzen.

116: Illegal use of function specifier.

Die Funktions-Spezifizierung (`"inline"` oder `"virtual"`) ist hier nicht zulässig.

117: "inline" used after definition or use of function.

Die Funktion wird hier erstmals als "inline" deklariert, wurde zuvor aber schon definiert oder aufgerufen.

118: Function redefined.

Der Anweisungsteil dieser Funktion wurde bereits definiert, oder sie wurde schon einmal als Amiga-Library-Funktion deklariert.

119: "inline" used with storage class specifier.

Eine `"inline"`-Funktion darf weder "extern" noch "static" deklariert sein.

120: Inconsistent linkage.

Die Speicherklasse einer Funktion oder eines Objekts wurde zuvor bereits anders deklariert, und die vorliegende neue Deklaration widerspricht dem.

121: Local function definitions can only be "extern".

In C++ dürfen Funktiosdeklarationen, die lokal in Funktionen auftreten, aus Gründen der Eindeutigkeit ausschließlich als "extern" definiert werden.

122: Syntax error.

Der wohl vielseitigste Fehler überhaupt - dem Compiler geht hier ziemlich die Orientierung ab.

123: Unknown linkage.

Maxon C++ kennt die Link-Spezifikationen "C", "C++", "Asm", "Pascal" und "Amiga", aber das, was hier steht, nicht.

124: Linkage specification may only occur in file scope.

Link-Spezifikationen dürfen nicht lokal in Funktionen, sondern nur auf Dateiebene stehen (Tutorial 2.2.1.2.3).

125: Pointer to member: Class expression expected.

Der Typ des linken Operanden von `".*"` bzw. `"->*"` stimmt nicht: Es muß sich um eine Klasse bzw. einen Zeiger darauf handeln.

126: Illegal pointer-to-member expression.

Typfehler: mindestens ein Operand von `".*"` bzw. `"->*"` hat einen unpassenden Typ.

127: Assignment: Left value must be L-Value.

Bei einer Wertzuweisung mit `"="` muß der linke Wert ein L-Wert sein - was denn auch sonst, denn schließlich kommt Begriff "L-Wert" auch genau daher (Tutorial 1.4.3.2, 1.4.3.11 et al.).

128: Type conflict.

Diese Typkonvertierung ist zumindest implizit nicht möglich.

129: Can't convert "qualified type*" to plain "type*".

Für einen Datentyp `"T"` kann ein Zeiger auf `"const T"` oder `"volatile T"` nicht in einen gewöhnlichen Zeiger auf `"T"` ungewandelt werden, denn das würde durchs Hintertürchen diese beiden Typqualifizierungen ad absurdum führen (Tutorial 2.2.3.1).

130: Access to private base class.

Es wurde, insbesondere bei einer impliziten oder expliziten Typumwandlung, auf eine private Basisklasse zugegriffen, wobei die allgemeinen Zugriffs-Kontroll-Regeln verletzt wurden.

131: Member function call: class expression has illegal type.

Beim Aufruf einer Member-Funktion ist der Ausdruck links vom `"."` bzw. `"->"` vom falschen Typ, z. B. `"const"` oder `"volatile"`.

133: Conversion "int" -> "enum": allowed in C mode only.

In C++ ist es nicht möglich, einen ganzzahligen Wert ohne Cast in einen Aufzählingstyp umzuwandeln (Tutorial 2.4.3).

134: Operator or type description expected.

Hinter dem Schlüssenwort `"operator"` sollte entweder ein überladbarer Operator (dann ist es eine Operator-Funktion) oder eine Typbeschreibung stehen (dann ist es eine Konvertierungsfunktion).

135: Conversion functions must be member of a class.

Da Konvertierungsfunktionen ein Klassenobjekt in einen anderen Datentyp umwandeln, sonst aber keine Parameter besitzen, können Sie ausschließlich als nichtstatische Member einer Klasse deklariert werden.

136: Conversion functions may not be static.

Eine Konvertierungsfunktion ist zwar eine relativ gewöhnliche Memberfunktion, darf aber nicht als statisch deklariert werden. Kein Wunder, schließlich soll sie ja ein Klassenelement in einen anderen Typ konvertieren und hat keine expliziten Parameter, und deshalb steht sie ohne den impliziten "this"-Parameter ziemlich arm da.

137: Ambiguous conversion function.

Die hier explizit genannte Konvertierungsfunktion gibt es in mehreren Basisklassen, weshalb sie nicht eindeutig ist. Diese Fehlermeldung ist im Grunde genommen analog zu Error #176 (`"<id1>"` defined in classes `"<id2>"` and `"<id3>"`), nur daß wir es hier nicht mit einem wirklichen Bezeichner, sondern dem Pseudo-Namen einer Konvertierungsfunktion zu tun haben.

138: Conversion function not found.

Die genannte Konvertierungsfunktion wurde nie zuvor deklariert.

139: Type conversion by ambiguous user conversion functions

Für die gewünschte Typkonvertierung stehen mehrere Konvertierungsfunktionen zur Verfügung, ohne daß klar ist, welche hier zu benutzen ist.

140: Operator overload: at least one parameter must be "class".

Wenn Sie einen Operator überladen, muß wenigstens einer der Operanden von irgendeinem Klassentyp sein.

141: Operator overload: Illegal number of formal parameters.

Die hier deklarierte Operatorfunktion hat zu viele oder zu wenige Parameter.

142: Operators may not have default arguments or ellipsis.

Operator-Funktionen dürfen keine Default-Argumente und erst recht keine Ellipse `"..."` haben.

143: Second parameter must be "int".

Bei dieser Operatorfunktion (z. B. `"++"` oder `"--"`) muß der optionale zweite Operand den Typ "int" haben.

144: Conversion functions may not have an explicit return type.

Der Ergebnistyp einer Konverierungsfunktion ist durch ihren Namen (operator <typbeschreibung>) bereits implizit definiert. Es darf bei der Deklaration sonst kein Ergebnistyp angegeben werden.

145: Operators must not be static members.

Wenn eine Operatorfunktion schon als Member deklariert wird, sollte sie nicht statisch sein. Wahrscheinlich stammen Sie aus Benhausen.

146: Unary operator "<op>" used on illegal argument.

Der genannte unäre Operator wird mit einem unzulässigen Operanden benutzt.

147: Binary operator "<op>" used on illegal arguments

Ein binärer Operator wurde auf Operanden angewandt, die auch unter Berücksichtigung aller Überladungen einfach nicht passen wollen.

148: "." and "->" need a class expression as left operand

Der Ausdruck links des Operators `"."` oder `"->"` muß eine Klasse, Struktur, Union oder etwas, was sich mittels einer Konvertierungsfunktion darin verwandeln läßt, sein. Beachten Sie bitte auch die Möglichkeit, daß der Fehler in Mißachtung der Bindungsregeln bestehen und sich durch ein zusätzliches Klammerpaar beheben lassen könnte.

149: This operator must be a member function.

Bestimmte Operatoren, z. B. `"="`, können auschließlich als Memberfunktionen überladen werden.

150: Empty source!

Der Quelltext ist, jedenfalls nachdem er den Proprozessor passiert hat, völlig leer - formal zwar erlaubt, im Zweifelsfall aber wohl eher Folge eines Versehens. Zumindest können wir uns sicher darauf einigen, daß ein leeres Programm kein sinnvolles ist.

151: Operator "sizeof": Type has undefined size.

Die hier mit `"sizeof"` erfragte Größe des Datentyps ist auch dem Compiler selbst bisher noch nicht bekannt.

152: "sizeof" needs an expression or a type in "()"'s

Der Ausdruck bzw. die Typbeschreibung hinter dem `"sizeof"` ist nicht korrekt (Tutorial 1.4.3.3).

153: Destructor: class name expected after "~".

In dem, was der Compiler hier für einen Destruktor-Namen hält, muß hinter der Tilde `"~"` der aktuelle Klassenname stehen.

154: Declaration: can not use a "( )" initializer here.

Der Compiler hat eine Initialisierung in Klammern `"( )"` gefunden (oder glaubt es zumindest), welche hier aus irgendwelchen Gründen, z. B. in einem `"typedef"` oder einer Funktionsdeklaration, nicht erlaubt ist.

155: No matching constructor.

Es gibt keinen passenden Konstruktor, der die vorliegende Klasse mit dem angegebenen Ausdruck initialisiert.

156: No default constructor for class "<name>".

Die genannte Klasse hat keinen Default-Konstruktor, weshalb hier eine Initialisierung erforderlich ist. Der Hinweis kann auch am Ende der Definition einer Klasse stehen, wenn sie einen Member oder eine Basisklasse eines solchen Typs, aber keinen Konstruktor hat.

157: Ambiguous: use constructor or user conversion?

Der Compiler kann sich hier nicht recht entscheiden, ob er bei einer Typkonvertierung von einem Konstruktor oder einer Konvertierungsfunktion Gebrauch machen soll (Tutorial 3.5.2, vorletztes Beispiellisting)

158: Destructor declaration: Empty parameterlist "()" expected.

Ein Destruktor darf keine Parameter haben, die leere Parameterliste muß aber trotzdem angegeben werden.

159: Destructor name does not match class name.

Ein Destruktor muß so heißen wie die Klasse, für die er definiert wird.

160: Name of baseclass or nonstatic member expected.

Innerhalb einer Konstruktor-Initialisierungs-Liste wird hier der Name einer Basisklasse oder eines nicht-statischen Members erwartet (Tutorial 3.3.3).

161: Ctor initializer: "(" expected.

In der Initialisierungsliste eines Konstruktors wird hier ein rundes Klammerpaar mit dem Initialisierungswert darin erwartet.

162: Initializer for Member "<name>" expected.

Der Member `<name>` benötigt eine Initialisierung, weil er z. B. eine Referenz, ein konstantes Objekt oder ein Klassenobjekt ohne Defautkonstruktor ist.

163: Member or baseclass initialized twice.

In einer Konstruktor-Initialisierungs-Liste taucht der Member bzw. die Basisklasse doppelt auf (Tutorial 3.3.3).

164: Member "<name>" not initialized.

Der Member `<name>` benötigt eine Initialisierung, weil er z. B. eine Referenz, ein konstantes Objekt oder ein Klassenobjekt ohne Defautkonstruktor ist.

165: Operator "new" must return a pointer.

Der Ergebnistyp eines überladenen `"new"`-Operators muß ein Zeigertyp sein - strenggenommen sogar ein Zeiger auf `"void"`, aber da ist MaxonC++ etwas toleranter als nötig (Tutorial 4.2.2.2.2).

166: Operator "delete" must return "void".

Überlädt man den Operator `"delete"`, muß er einen Ergebnistyp haben, und zwar `"void"` und sonst nichts (Tutorial 4.2.2.2.3).

167: Operator "new": type expected.

Hinter `"new"` wird ein Datentyp erwartet (Tutorial 2.6.2).

168: This type does not represent an object.

Mit `"new"` dürfen nur Objekte erzeugt werden - und z. B. Referenzen, Funktionen oder "void" sind in diesem Sinne keine.

169: Vector element type has undefined size.

Der Datentyp der Vektorelemente (und damit auch der Vektortyp selbst) hat einen Speicherplatzbedarf, der dem Compiler momentan nicht bekannt ist.

170: Virtual functions differ in return type.

Es wurde versucht, mit `"new"` ein Objekt eines Typs bisher unbekannter Größe zu erzeugen.

171: Allocation of vectors: no initializer allowed.

Wird mit `"new"` ein Vektor erzeugt, kann dieser nicht explizit initialisiert werden. Auch bei Vektoren von Klassen findet nur eine implizite Initilisierung durch den Default-Konstruktor statt.

172: Operator "delete": 1st parameter must have type "void *".

Der erste Parameter eines überladenen Operators `"delete"` kann ausschließlich vom Typ `"void*"` sein (Tutorial 4.2.2.2.3).

173: Operator "delete": 2nd parameter must have type "size_t".

Wenn ein `"delete"`-Operator zwei Parameter hat, muß der zweite den Typ `"size_t"`, und das ist in Maxon C++ "unsigned int", haben (Tutorial 4.2.2.2.3).

174: Operator "delete": too many parameters

Mehr als zwei Parameter darf der `"delete"`-Operator nicht haben (Tutorial 4.2.2.2.3).

175: Can not overload operator "delete".

In jeder Klasse kann nur ein `"delete"`-Operator definiert sein (Tutorial 4.2.2.2.3).

176: "<id1>" defined in classes "<id2>" and "<id3>".

Der Member `<id1>` tritt in den sich gegenseitig nicht überdeckenden Basisklassen `<id2>` und `<id3>` der vorliegenden Klasse auf und kann deshalb nicht eindeutig zugeordnet werden.

177: Virtual functions must be members of a class.

Natürlich können virtuelle Funktionen nur Memberfunktionen einer Klasse sein.

178: Virtual functions can not be static.

Eine virtuelle Memberfunktion ist naturgemäß an ein ganz bestimmtes Objekt gebunden und kann deshalb niemals als "static" deklariert werden.

179: Class "<id1>" is abstract (pure virtual function "<id2>" in baseclass "<id3>").

`<id1>` ist eine abstrakte Klasse und kann als solche hier nicht benutzt werden (insbesondere dürfen auf keine Weise Objekte dieser Klasse definiert werden). Als besonderer Service wird darauf hingewiesen, daß `<id2>` eine der rein virtuellen Funktionen der ist und ursprünglich in der Basisklasse `<id3>` definiert wurde.

180: Virtual functions differ in return type.

Die Funktion überdeckt eine gleichnamige virtuelle Funktion in einer Basisklasse, hat aber einen anderen Ergebnistyp.

181: Class "<id1>" is abstract (pure virtual function "<id2>").

`<id1>` ist eine abstrakte Klasse und kann als solche hier nicht benutzt werden (insbesondere dürfen auf keine Weise Objekte dieser Klasse definiert werden). Als besonderer Service wird darauf hingewiesen, daß `<id2>` eine der rein virtuellen Funktionen der Klasse ist.

182: Constructor initializer: "{" expected.

Am Ende der Initialisierungsliste eines Konstruktors wird der mit `"{"` beginnende eigentliche Funktionsrumpf erwartet (Tutorial 3.3.3).

183: Calling pure virtual function.

Es ist schlechterdings unmöglich (und führt zu einem Linker-Fehler), eine als rein virtuell definierte Memberfunktion aufzurufen (Tutorial 3.2.3.3). Rein formal kann man es aber trotzdem versuchen, und zwar über einen qualifizierten Bezeichner oder innerhalb eines Konstruktors. Mit dieser Fehlermeldung haut der Compiler einem dann auf die Pfoten.

184: Pointer expression expected.

Hinter `"delete"` muß als Operand natürlich ein Zeigerausdruck stehen.

185: Can´t convert virtual base class to derived class.

Hat man einen Zeiger auf eine virtuelle Basisklasse, kann man ihn auch durch einen Cast nicht in einen Zeiger auf eine abgeleitete Klasse umwandeln, denn in diese Richtung gibt es keinen Zeiger (Tutorial 3.2.5, Beispiel mit `"VirtBasis"` und `"VirtAbgeleitet"`).

186: Reference conversion: L-value expected.

Die Referenz muß hier mit einem geeigneten L-Wert initialisiert werden. Der Compiler kann hier also kein temporäres Objekt einführen.

187: Can not subtract pointers of different types.

Die mit "-" berechnete Differenz zweier Zeigerausdrücke ist nicht etwa die Differenz der Speicheradressen, sondern die Anzahl der dazwischen liegenden Vektorelemente. Deshalb können ausschließlich Zeiger auf den selben Typ voneinander subtrahiert werden.

188: Referenced type has unknown size.

Bei Additionen und Subtraktionen auf Zeigern (auch mit "++", "--", "+=", "-=") muß die Größe des vom Zeiger referierten Datentyps bekannt sein. Bei Zeigern auf Klassen und Strukturen muß diese z. B. definiert worden sein.

189: Division by zero.

In einem konstanten Ausdruck, der zur Compilezeit ausgewertet wurde, trat eine Division durch Null auf.

190: Operator requires integral operands.

Bei "<<", ">", "%" und anderen Operatoren sind keine Fließkomma-, sondern nur ganzzahlige Operanden erlaubt.

191: Attempt to assign to const object.

Sinn und Zweck eines als "const" deklarierten Objekts ist, daß ihm eben nach der Initailisierung kein neuer Wert mehr zugewiesen werden darf - und das wurde auf die eine oder andere Weise versucht (Tutorial 2.2.3.1).

192: No assignment to vectors, functions, and void expressions.

Vektoren, Funktionen und "void"-Ausdrücke können grundsätzlich nicht auf der linken Seite einer Wertzuweisung stehen.

193: Expressions left and right of ":" need common type.

Bei einem bedingten Ausdruck (Tutorial 1.4.3.10) müssen der "if"- und der "else"-Teilausdruck (also die Ausdrücke beiderseits des ":") einen gemeinsamen Datentyp haben oder sich zumindest ineinander umwandeln lassen können.

194: Conditional expression: ":" expected.

Ein bedingter Ausdruck erfordert hinter dem "?" stets gleichermaßen einen "if"- und einen "else"-Teil, die durch ein ":" zu trennen sind.

195: Condition expected.

Hier wurde eine logische Bedingung erwartet. Dabei kann es sich um einen beliebigen skalaren Ausdruck handeln.

196: Illegal parameter for "operator = ()".

Einem Klassenobjekt, auf dem der Operator "=" überladen ist, wird hier ein unpassender Ausdruck zugewiesen.

197: += / -= used on pointer: integral expr. as 2nd operand required.

Bei den Operatoren `"+="` und `"-="` mit einem Zeigerausdruck als linkem Operanden muß der rechte Operand (also das, was addiert werden soll) natürlich ganzzahlig sein.

198: Attempt to assign to undefined class.

Eine Struktur oder Klasse wurde zwar deklariert, z. B. `"class Hallo;"`, aber ihr Inhalt (=ihre Member) noch nicht definiert. Deshalb ist noch keine Wertzuweisung zwischen Objekten dieser Klasse möglich.

199: Label "<name>" redefined.

Ein Sprunglabel namens `<name>` gibt es in dieser Funktion schon.

200: Label expected.

Hinter `"goto"` wird ein Label erwartet, und das ist in C grundsätzlich ein Bezeichner.

201: Label "<name>" undefined.

Die Sprungmarke `<name>` wurde benutzt, aber nirgendwo gesetzt.

202: Jump past initialization of object "<name>".

Eine Sprunganweisung, insbesondere bei `"goto"` oder `"switch"`, aber unter bestimmten Umständen auch in Form einer Schleifenstruktur oder Fallunterscheidung, überspringt die Initialisierung der Variablen `<name>`.

203: "(" expected.

Hinter `"for"`, `"switch"` oder einem ähnlichen Schlüsselwort wird ein Klammernpaar erwartet.

204: "else" without "if".

Ein `"else"` ohne passendes `"if"` wurde gelesen.

205: "continue" used outside iteration statement.

Die Anweisung `"continue"` kann nur innerhalb einer geeigneten Schleifenanweisung gesetzt werden.

206: "break" without matching iteration or switch statement.

Die `"break"`-Anweisung darf nur innerhalb von geeigneten Schleifen oder `"switch"`-Statements auftreten (Tutorial 1.5.4.2).

207: "while" expected.

Am Ende der `"do"`-Schleife wird hier das `"while"` schmerzlich vermißt.

208: Function needs return value.

Da die Funktion einen Ergebnistyp ungleich `"void"` hat, wird hinter dem `"return"` ein passender Rückgabeausdruck erwartet.

209: Void functions may not return data.

Die Funktion hat den Ergebnistyp `"void"`, weshalb hinter `"return"` keinerlei Ergebnistyp (auch kein `"void"`-Ausdruck) stehen darf.

210: Expression does not match function return type.

Der Ausdruck hinter dem `"return"` paßt typmäßig nicht zum Ergebnistyp der Funktion und kann auch nicht dorthin umgewandelt werden.

211: Integral or enumerated expression expected.

Ein Ausdruck eines ganzzahligen oder Aufzählingstyps wird erwartet, insbesondere als Argument von `"switch"`.

212: "case" without "switch"

Ein `"case"`-Label ohne passendes `"switch"`-Statement wurde gefunden.

213: ":" expected.

Hinter einer `"case"`-Sprungmarke fehlt hier der Doppelpunkt.

214: Expression does not match type of "switch" expression.

Der Ausdruck hinter dem `"case"` läßt sich (jedenfalls implizit) nicht in den (ganzzahligen oder Aufzählungs-) Typ des Ausdrucks der `"switch"`-Anweisung umwandeln.

215: "case" requires constant expression.

Naturgemäß darf hinter einem `"case"`-Label ausschließlich ein konstanter skalarer Ausdruck stehen.

216: Constant appears twice in same "switch" statement.

Jeder konstante `"case"`-Wert darf innerhalb eines `"switch"` nur genau einmal auftreten.

217: "default" without "switch"

Genau wie `"case"` kann `"default"` nur innerhalb einer `"switch"`-Anweisung auftreten.

218: Each "switch" statement may have only one "default" label.

In jeder `"switch"`-Anweisung darf das Label `"default:"` höchstens einmal auftreten.

219: Old-style parameter declaration expected.

Innerhalb einer Parameterlistendefinition im alten Kernighan-Ritchie-Stil liegt hier ein (meist offensichtlicher) syntaktischer oder semantischer Fehler vor.

220: Can not overload old-style functions.

MaxonC++ weigert sich, bei Funktionen mit Parameterlisten im alten K&R-Stil das C++-Feature Überladung zuzulassen. Das ist ganz einfach zu blöd.

221: Can not take address of register parameter.

Wurde ein Parameter in MaxonC++ als `"register"` deklariert, darf seine Adresse weder direkt (mit dem unären `"&"`-Operator) noch indirekt (durch Setzen einer Referenz auf die Variable) genommen werden. Das ist auch einsichtig, denn es wurde zuvor der gesamten Umwelt groß und breit kundgetan, daß der Parameter beim Funktionsaufruf in ein Register geladen werden soll, und deshalb kann der Compiler die Variable schlecht noch nachträglich auf den Stack raufschieben.

222: Inconsistent use of register parameters.

Zwei Prototypen für ein und dieselbe Funktion unterscheiden sich hinsichtlich der Definition der Registerparameter.

223: Sorry, can't convert virtual base class to derived class.

Ein Zeiger auf eine virtuelle Basisklasse kann nicht wieder in einen Zeiger auf die abgeleitete Klasse umgewandelt werden (Tutorial 3.2.5).

224: Function "<xxx>" has internal linkage and is undefined.

Die Funktion `<ident>` wurde als `"static"` deklariert, aber nirgendwo im Programm definiert. Da sie offenbar nicht über den Linker eingebunden werden kann, ist dies ein Fehler.

225: Name of pointer/int variable expected.

Innerhalb einer Amiga-Linkspezifikation wird hier der Name der Basisvariablen, die (sofern überhaupt schon deklariert) bitteschön `"int"` oder von einem Zeigertyp sein sollte (Tutorial 2.2.1.2.4).

226: Library vector Offset ("= <short constant>") expected.

Bei der Linkspezifikation `"Amiga"` wird hinter jeder Funktionsdeklaration ein `"="` und der ganzzahlige Offset erwartet (Tutorial 2.2.1.2.4).

227: Implicit conversion "void*" -> "TYPE*": allowed in C Mode only.

Nur in ANSI C kann ein Zeiger auf `"void"` implizit (d. h. ohne Cast und dergleichen) in einen anderen Zeigertypen umgewandelt werden. C++ erwartet hier eine explizite Typkonvertierung (Tutorial 2.1.1.4).

228: Can not mix declarations and other statements in C mode.

In ANSI C müssen innerhalb eines Blocks sämtliche Variablendeklarationen vor der ersten Anweisung stehen (Tutorial 1.5.2). Das "other" steht hier deshalb, weil eine Deklaration in C++ ebenfalls als Anweisung gilt.

229: Sorry, but ANSI C does not know member functions...

In ANSI C gibt es definitiv keine Member-Funktionen.

230: Syntax error in #pragma

In einem MaxonC++ bekannten `#pragma` wurde ein Fehler gefunden.

231: "amicall": non-overloaded function expected.

Die `"#pragma amicall"`-Deklaration darf sich nur auf Funktionen beziehen, für die bereits ein Prototyp gelesen wurde, und ist für überladene Funktionen auch nicht erlaubt (Tutorial 5.5.3).

232: "amicall": int/pointer variable expected.

Innerhalb eines `"amicall"`-Pragmas wird hier die Basisvariable, nämlich ein Bezeichner für eine globale `"int"`- oder Pointervariable, erwartet (Tutorial 5.5.3).

233: Function already used.

Für eine Funktion, die zuvor bereits ein "gewöhnlicher" Aufruf stattgefunden hat, wird hier nachträglich ein `"#pragma amicall"` angegeben, was nicht erlaubt ist.

234: Every reference needs an initializer.

Referenz-Objekte müssen grundsätzlich initialisiert werden.

235: Can not take address of a "register" object.

Es ist in C nicht erlaubt, die Adresse eines als `"register"` deklarierten Objekts zu nehmen, nicht einmal dann, wenn das Objekt sowieso nicht in ein Register geladen wird. C++ ist da toleranter: wenn die Adresse der Variablen irgendwo auftaucht, wird die Variable eben nicht in einem Register abgelegt. Mit "Adresse nehmen" ist insbesondere die Anwendung des unären Operators `"&"` gemeint, aber auch implizite Adressermittling, z. B. wenn eine Referenz mit dem Objekt initialisiert wird (Tutorial 2.2.1.3).

236: Syntax error inside inlined member function "<xxx>".

Inline-Memberfunktionen, also solche kurzen Funktionen, die direkt in der Klassendefinition selbst definiert werden, werden zunächst nur sehr grob (durch Zählen der Klammern) analysiert, um erst nach Ende der Klassendefinition endgültig geprüft zu werden. Bei dieser ersten groben Analyse wurde ein offensichtlicher syntaktischer Fehler gefunden.

237: Unexpected end of source.

Der Quelltext endet vorzeitig, z. B. mitten in einer Funktions- oder Klassendefinition.

238: Automatic variable used outside of its function.

In C++ ist es tatsächlich möglich, Funktionen ineinander zu verschachteln, indem man nämlich lokale Klassen mit Memberfunktionen deklariert. In diesen Memberfunktionen sind auch die Bezeichner der umschließenden Fnktion sichtbar - aber die Variablen dürfen trotzdem nicht angerührt werden, weil dies enorme implementatorische Schwierigkeiten verursachen würde.

239: Can't reference "const TYPE" with plain "TYPE &".

Diese Fehlermeldung ist im Grunde genommen das Gegenstück zu Error #129 (Can't convert "qualified type*" to plain "type*"), nur mit Referenzen statt Zeigern.

240: A template declaration may only appear as a global declaration.

Templates sind generell global. Sie dürfen nur auf Dateiebene, also keinesfalls in einer Klasse oder Funktion, deklariert und definiert werden.

241: Operator "new": first operand must be of type "size_t".

Der erste Parameter einer `"new"`-Operatorfunktion muß vom Typ `"size_t"`, also `"unsigned int"` in der vorliegenden Implementierung, sein (Tutorial 4.2.2.2.2).

242: Template argument list: "<" expected.

Die Argumentliste eines Templates beginnt stets mit der öffnenden spitzen Klammer `"<"`, und genau dies wird hier vermißt.

243: Template argument list: ">" expected.

Das Ende einer Template-Argumentliste bildet die schließende Klammer. Der Compiler ist der Überzeugung, daß die Argumentliste jetzt zu Ende ist, und erwartet deshalb das `">"`.

244: Template argument expected.

In der Template-Deklaration wird hier ein Typ- oder Wertparameter erwartet.

245: Template argument list differs from previous declaration.

Templates sind nicht (wie Funktionsnamen) überladbar. Folglich muß die Parameterliste eines Templates bei jeder Deklaration oder Definition praktisch identisch sein. Allenfalls die Namen der Argumente dürfen differieren.

246: This template has already been defined.

Es wird soeben versucht, ein bereits prima definiertes Template noch einmal zu definieren.

247: Syntax error inside class template "<xxx>".

Klassentemplates werden zunächst nur sehr grob auf syntaktische Korrektheit geprüft (genaugenommen werden hauptsächlich die Klammern gezählt), und die eigentliche syntaktische und semantische Überprüfung erfolgt erst, wenn das Template benutzt wird. Hier wurde aber schon bei der ersten Prüfung ein Fehler entdeckt.

248: Type argument expected.

Als Argument (bei einer Templatebenutzung) wird an dieser Stelle eine Typbeschreibung erwartet.

249: Template argument list: "," expected.

Die Argumentliste des Templates ist noch keineswegs zuende. Deshalb werden ein Komma und noch mindestens ein weiteres Argument erwartet.

250: Constant expression as template argument expected.

Wenn ein Template ein Wert-Argument erwartet, muß dieses auch konstant sein. Pointer auf statische Objekte gehen notfalls aber auch durch.

251: Template: class or function declaration expected.

Templates gibt es in C++ für Klassen und Funktionen und sonst für nix.

252: Syntax error inside template function "<name>".

Funktionstemplates werden zunächst nur sehr grob auf syntaktische Korrektheit geprüft (genaugenommen werden hauptsächlich die Klammern gezählt), und die eigentliche syntaktische und semantische Überprüfung erfolgt erst, wenn aus dem Template eine Funktion erzeugt wird. Hier wurde aber schon bei der ersten Prüfung ein Fehler entdeckt.

253: Function templates: Argument "<name>" never used.

Alle Parameter (und das sind stets Typparameter) eines Funktionstemplates müssen aus Gründen der Eindeutigkeit in der Parameterliste der Funktion auftauchen. Ein Auftreten im Ergebnistyp oder Anweisungsteil der Funktion zählt in diesem Zusammenhang nicht.

254: No such member function in class "<name>".

Die hier deklarierte Member-Funktion tritt in der genannten Template-Klasse nicht auf. Wahrscheinlich gibt es sie sogar in überhaupt keiner aus diesem Template generierten Klasse.

255: Template function redefined.

Das Funktionstemplate war zuletzt mit anderen Parametern oder einem anderen Ergebnistyp deklariert worden.

256: Arguments do not match function template.

Bei der Benutzung eines Templates passen die hinter dem Templatenamen in spitzen Klammern `"<...>"` angegebenen Argumente nicht.

257: Class name collides with template name.

Auf Dateiebene oder an sonstwie unpassender Stelle trat eine Typangabe wie `"class XXX"` auf, wobei `"XXX"` Name eines Klassentemplates ist, aber keine Template-Argumentliste folgt. Dies ist nicht zulässig.

258: Inconsistent exception specification

Wenn ein Prototyp einer Funktion mehrfach aufgeführt wird, darf die Ausnahme-Deklaration jedesmal wiederholt werden - man kann es aber auch lassen. Wenn man sich die Mühe macht, müssen die Deklarationen aber natürlich übereinstimmen.

259: "{" expected.

Try-Blöcke und Catch-Handler sind stets Verbundanweisungen und müssen deshalb in geschweifte Klammern eingeschlossen werden.

260: "catch" expected.

Hinter einem Try-Block muß mindestens eine Catch-Answeidsung folgen. Sonst hätte das "try" ja auch ziemlich wenig Sinn.

261: "throw": Expression expected.

Hinter `"throw"` kann entweder garnichts oder ein Ausdruck stehen. Eine Typangabe z. B. reicht nicht.

262: Illegal jump into try block.

Mit einem `"goto"`-Sprung wurde in einen Try-Block hineingesprungen, was der Compiler nicht erlaubt.

263: No "#pragma header" found.

Wenn Sie mit der Option `"-h"` bzw. der entsprechenden Einstellung in einem Dialogfenster eine vorcompilierte Header-Datei benutzen wollen, muß Ihr Quelltext eine Zeile `"#pragma header"` enthalten. Diese muß in der Haupt-Quelldatei und nicht etwa in einer Includedatei stehen.

264: Can not open header file "<name>".

Die vorcompilierte Header-Datei kann nicht geöffnet werden. Diese Meldung kann beim Lesen oder Schreiben der Datei auftreten.

265: Header file is invalid!

Das, was mit der Option `"-h"` bzw. der entsprechenden Einstellung in einem Dialogfenster als Header-Datei geladen werde sollte, ist gar keine.

266: Header file has wrong version number - please recompile it!

Die vorcompilierte Headerdatei konnte zwar geladen werden, aber sie wurde mit einer anderen Compilerversion erzeugt und paßt deshalb nicht zum momentan benutzten Compiler.

267: "#pragma header" may appear on file scope only.

Die Zeile `"#pragma header"` darf nur zwischen den Deklarationen auf Datei-Ebene auftreten, also insbesondere nicht innerhalb von Funktions- oder Klassendefinitionen und auch nicht mitten in einer globalen Deklaration.

B-3. Sonstige Fehlermeldungen

Nachdem der Compiler den Analyseteil abgeschlossen und `"No errors"` gemeldet hat, ist aber noch nicht alles gelaufen, was potentiell schiefgehen kann. Da gibt es zum Beispiel theoretisch die folgende Meldung:

"Internal error in Code generation"

Ich hoffe sehr, daß dieser Fehler bei Ihnen niemals auftritt. Da MaxonC++ paranoid programmiert wurde, prüft es an gegebener Stelle öfters, ob das, was es da macht, überhaupt so sein kann. Kommt es dabei zu dem Schluß, daß intern irgendetwas höllisch schiefgelaufen sein muß,

macht es sich benerkbar - im Analyseteil mit der Fehlermeldung `"Compiler is in Panic!"`, während der Codeerzeugung mit "Internal Error...".

Was tun, wenn so ein Fehler trotzdem auftritt? Melden Sie sich umgehend bei MAXON, am besten mit einer möglichst detaillierten Fehlerbeschreibung, denn es handelt sich hier um einen Bug des Compilers und liegt nicht unbedingt an Ihrem Programm, auch wenn die Chancen gut stehen, daß ein solcher Fehler durch ein Übermaß an Fehlermeldungen provoziert wurde.

Als "erste Hilfe" können Sie versuchen, herauszufinden, welche Stelle Ihres Programms die Ursache war: Lassen Sie sich mit der Option `"-as"` bzw. dem ***"Source in Asm"***-Gadget ein kommentiertes Assemblerlisting erzeugen, und sehen Sie darin nach, an welcher Quelltextposition die Codererzeugung abgebrochen wird (da endet das Assembler-Listing). Vielleicht fällt Ihnen ja etwas ein, wie Sie die betreffende Stelle unverfänglicher programmieren können.

Wie gesagt, ich hoffe, daß der oben genannte Fehler bei Ihnen wirklich niemals unverschuldet auftritt. Eine zweite Fehlermeldung, die ebenfalls während der Codegenerierung auftreten kann, ist weit weniger problematisch:

"Workspace overflow."

Diese Worte wollen Ihnen ganz einfach sagen, daß der Compiler-Arbeitsspeicher zu knapp bemessen ist. Erhöhen Sie diese Einstellung einfach mit der Option `"-w"` bzw. dem entsprechenden Gadget im Compiler-Requester nach Gutdünken und starten Sie den Compiler erneut.

Maxon C++ 3

Features

Light-Version
- C++-Compiler (AT&T 2.0 Standard)
- ANSI C-Compiler (Aztec- und Lattice-ANSI-kompatibel)
- CLI-Compiler
- HotHelp 3-Reader zum Anzeigen von HotHelp-Projekten

Developer-Version
- C++-Compiler (AT&T 3.0 Standard)
- Templates, Exceptions
- ANSI C-Compiler (wie Light-Version)
- CLI-Compiler
- beinhaltet zusätzlich: MaxonHotHelp 3, MaxonSDB, MaxonEASY-OBJECTS und MaxonASM-CLI

Compiler (C/C++)
- komfortable Projektverwaltung (Make)
- direkter Fehlerstellenansprung
- voll in den Editor integriert
- über ARexx-Schnittstelle steuerbar
- sehr schnelle Turnaroundzeiten
- echte vorkompilierte Include-Dateien (Preincludes)
- Include-Dateien für Kickstart 3.1
- speicherbarer Programmstatus
- Codegenerator für prozessorspezifischen Code 68010/20/30 und 68881/2 FPU

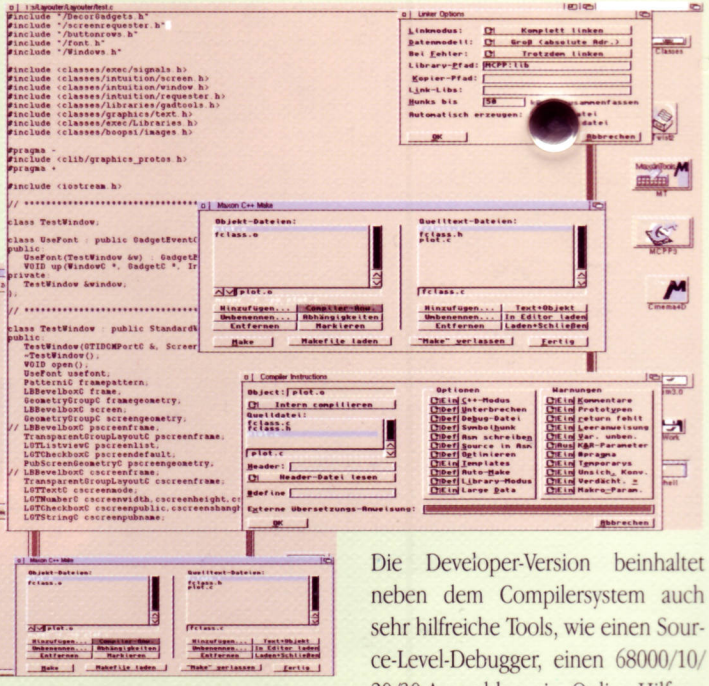
Programmeditor (EDWARD)
- Multi-Window-Technik
- beliebig viele Texte gleichzeitig bearbeitbar (speicherabhängig)
- Tastatur- und Menüfunktionen frei belegbar
- speicherbarer Programmstatus
- ARexx-Schnittstelle

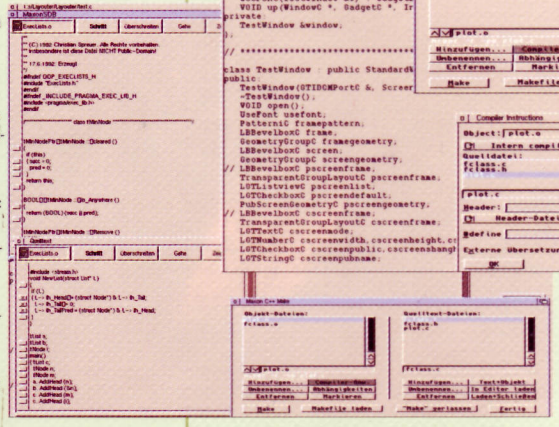
Hilfesystem HotHelp 3
(nur in der Developer-Version)
- Deutschsprachige Beschreibung der Library- und Device-Funktionen zu Amiga-OS 2.1
- jederzeit aktivierbar
- kontextsensitiv
- komfortabler Editor für eigene Projekte
- Online-Projekte (direkte übernahme von Texten und Sourcecode in das Hilfesystem)
- ARexx-Schnittstelle
- Grafikeinbindung direkt im Hilfetext
- direktes Lesen von AmigaGuide™-Projekten
- lizenzfreie Weitergabe des HotHelp 3-Anzeigers

Features der Klassenbibliothek (MaxonEASY-OBJECTS)
(nur in der Developer-Version)
- Ressourcenhandling: Betriebssystemressourcen (Speicher, Signale, Fenster, Bildschirme etc) werden automatisch verwaltet und bei Programmende freigegeben (keine aufwendige Fehlerbehandlung mehr nötig).
- Fehlerbehandlung durch Exceptions
- Handlerkonzept
- Datenstrukturen: Listen, dynamisch große Arrays, Datenpuffer (z.B. für I/O-Operationen)
- Verwaltung folgender Betiebssystemresourcen: Exec: Signale, Nachrichtenports, Speicher, Libraries, Devices, Listen - DOS: Locks, Dateien, Pfadnamen, Kommandozeilenargumente, Notifikation - Utility: Hooks und Taglisten - Intuition: Bildschirme, Fenster, IDCMP-Nachrichten, Systemrequester - Libraries: GadTools, ASL, Icons, Workbench, Tooltypes, Applikationenfenster
- BOOPSI Unterstützung
- automatisch größenveränderbare und font-veränderbare Fenster durch den Layouter
- deutschsprachige Online-Dokumentation

Externer Assembler (MaxonASM-CLI)
(nur in der Developer-Version)
- 68000/10/20/30 Makro-Assembler
- FPU 68881/2 und MMU-Unterstützung
- Multi-Pass-Optimierung erzeugt sehr schnellen und kompakten Programmcode.
- voll in die Projektverwaltung integrierbar

C/C++ Source Level Debugger (MaxonSDB)
(nur in der Developer-Version)
- Multi-Window-Technik
- einfache Bedienung - Drag&Drop
- Beobachten von Variableninhalten
- Haltepunkte
- direkt vom Compiler aus aufrufbar





MaxonC++ ist die erste konsequente Implementierung des C++ 3 Standards auf dem Amiga.

Das MaxonC++-Compilersystem beinhaltet sowohl einen ANSI C-, als auch einen zum AT&T 3.0-Standard kompatiblen C++-Compiler. Der Compiler erzeugt aus den C++-Quelltexten direkt ausführbare Programme ohne Umweg über C, Assembler oder andere Sprachen. Durch die volle Systemintegration werden überzeugend schnelle Turnaroundzeiten erreicht.

Die Developer-Version beinhaltet neben dem Compilersystem auch sehr hilfreiche Tools, wie einen Source-Level-Debugger, einen 68000/10/20/30-Assembler, ein Online-Hilfesystem mit der deutschsprachigen Beschreibung der Systemfunktionen und eine umfangreiche Klassenbibliothek.

Das auch in der Light-Version aufwendige und gut strukturierte deutschsprachige Handbuch ist ein kompletter ANSI-C/C++-Programmierlehrgang und auch für Einsteiger zu empfehlen.

Lauffähig auf allen Amigas mit min. 2 MB Speicher und Amiga-OS 2.0. Bei Nutzung der Klassenbibliothek werden 6 MB Speicher benötigt.

AMIGA®

4 010509 411302

MAXON Computer GmbH • Industriestr. 26
D-65760 Eschborn • Telefon 06196/481811
Irrtümer und Änderungen vorbehalten!

MAXON computer